Maria Inês Vasconcellos Furtado
José Cláudio Garcia Damaso
Lúcio Pereira de Andrade

PROGRAMAR C

Autores:
Maria Inês Vasconcellos Furtado
José Cláudio Garcia Damaso
Lúcio Pereira de Andrade

Edição: 1ª, 2025
DOI: 10.5281/zenodo.16811982
ISBN: 978-989-33-8373-5

Licença: CC BY 4.0 (*Creative Commons Attribution International)*
https://creativecommons.org/licenses/by/4.0/

Lisboa, Portugal: Zenodo, 2025

OS AUTORES

Este trabalho é fruto da colaboração entre três pessoas que, ao longo dos anos, fizeram da convivência acadêmica uma parceria marcada pela amizade, respeito e sinergia intelectual.

Inês, Cláudio e Lúcio compartilham não apenas a trajetória docente e a paixão pelo ensino e pesquisa, mas também o compromisso com uma universidade viva, conectada com as demandas sociais e aberta ao diálogo entre saberes.

Cada um, com sua formação e experiências singulares, das engenharias e sistemas computacionais às ciências contábeis e à gestão educacional, contribui com olhares complementares que enriquecem a abordagem dos temas aqui tratados. Juntos, vêm desenvolvendo projetos interdisciplinares que integram teoria e prática, ensino e extensão, técnica e sensibilidade.

Mais do que resultados acadêmicos, o que se percebe neste trabalho é o reflexo de uma convivência marcada pela escuta atenta, pelo estímulo à criatividade e pela construção conjunta de ideias.

PREFÁCIO

Este livro advém da vontade dos autores de oferecer aos estudantes, profissionais e entusiastas da programação um material sólido, claro e acessível para aprender a linguagem C.

Mais do que apresentar conceitos técnicos, o objetivo é criar uma ponte entre a teoria e a prática, fornecendo exemplos comentados, exercícios graduais e explicações para facilitar a compreensão mesmo para quem nunca programou.

A linguagem C é a base de muitos sistemas e linguagens modernas e dominá-la é uma abertura para o desenvolvimento do pensamento computacional estruturado e para a programação eficiente. Ao longo dos capítulos, o leitor encontrará desde fundamentos como tipos de dados e estruturas de controlo até tópicos mais avançados, como manipulação de ficheiros, ponteiros e estruturas dinâmicas.

Este livro foi pensado objetivando o ensino da linguagem C de forma progressiva, permitindo que o leitor avance no seu próprio ritmo. Os exemplos práticos oferecidos podem ser adaptados para diferentes contextos, estimulando o raciocínio lógico e a autonomia do seu leitor na resolução de problemas.

É relevante destacar que este manual se destina a estudantes de cursos técnicos e universitários que necessitam aprender C para disciplinas de programação, algoritmia ou estruturas de dados. São também contemplados profissionais que pretendem reforçar as suas bases de programação e autodidatas que desejam aprender a programar de forma estruturada e consistente.

Como recomendação de estudo, cada capítulo deve ser lido de forma sequencial, uma vez que os conceitos são introduzidos gradualmente. Os exemplos de código devem ser digitados e executados pelo leitor, pois a prática é essencial para consolidar o conhecimento. Os exercícios propostos ao final de cada capítulo servem como um desafio para testar a compreensão e podem ser adaptados para projetos próprios.

Espera-se que esta obra inspire, ensine e motive cada leitor a construir uma base sólida em programação e a explorar o vasto universo da informática com confiança e criatividade.

Lisboa, agosto de 2025

Silvana dos Santos Ambrosoli

Para Millan

Índice

1 CONCEITOS BÁSICOS DE PROGRAMAÇÃO 13

1.1 UMA BREVE HISTÓRIA DA PROGRAMAÇÃO 13
1.1.1 O Computador como Máquina de Processar Informação 13
1.2 POR QUE APRENDER A LINGUAGEM C 15
1.2.1 Representação de Algoritmos 16
1.3 EXERCÍCIO RESOLVIDO 20
1.4 PRONTO A PROGRAMAR NA LINGUAGEM C? 21
1.4.1 Um programa C começa com uma estrutura mínima 22
1.4.2 Formatação livre e indentação 23

2 TIPOS DE DADOS E VARIÁVEIS 26

2.1 IMPORTÂNCIA DOS TIPOS DE DADOS 26
2.2 TIPOS PRIMITIVOS DE DADOS 27
2.2.1 Atribuição, Inicialização e Inicialização Composta 28
2.2.2 Atribuição simples 28
2.2.3 Inicialização 29
2.2.4 Inicialização composta 29
2.2.5 Atribuições encadeadas 30
2.2.6 Exemplo ilustrativo 30
2.3 MODIFICADORES DE TIPO 32
2.4 DECLARAÇÃO DE VARIÁVEIS 34
2.4.1 Regras básicas para nomeação de variáveis em C: 35
2.5 INICIALIZAÇÃO DE VARIÁVEIS 37
2.6 CONSTANTES 38
2.7 COMO TROCAR O CONTEÚDO DE DUAS VARIÁVEIS 41
2.8 CONSTANTES SIMBÓLICAS E DIRETIVAS DE PRÉ-PROCESSADOR 42
2.9 EXERCÍCIOS PROPOSTOS 43

3 OPERADORES 45

3.1 OPERADORES ARITMÉTICOS 45
3.2 OPERADORES RELACIONAIS 46
3.3 OPERADORES LÓGICOS 48
3.4 OPERADORES DE ATRIBUIÇÃO COMPOSTA 50
3.5 OPERADORES DE INCREMENTO E DECREMENTO 52
3.6 PRECEDÊNCIA DE OPERADORES 53

4 ENTRADA E SAÍDA DE DADOS 58

4.1 BIBLIOTECA PADRÃO STDIO.H 58

4.2 Especificadores de Formato 61
4.3 Comentários 63
4.4 Caracteres especiais (escape) em printf() 63
4.5 Saída de Dados com printf() 65
4.6 Formatação de casas decimais 66
4.7 Entrada de Dados com scanf() 66
4.7.1 Leitura de Dados de uma String com sscanf() *68*
4.8 Exercícios Propostos 71

5 ESTRUTURAS CONDICIONAIS 74

5.1 Introdução às Estruturas Condicionais 74
5.2 Instrução if() 74
5.3 Instrução if() ... else 75
5.4 Instruções Encadeadas (else if()) 76
5.5 Instrução switch() case 77
5.6 Operador Ternário (? :) 79
5.7 Exercícios Propostos 80

6 ESTRUTURAS DE REPETIÇÃO 82

6.1 Repetir bloco de comandos em C 82
6.2 Principais tipos em C 82
6.3 Estrutura while() 83
6.4 Estrutura do...while (); 84
6.5 Estrutura for() 85
6.6 Comandos de Controlo 89
6.6.1 break *89*
6.6.2 continue *89*
6.7 Repetições Aninhadas 90
6.8 Exercícios Propostos 92

7 ARRAYS 96

7.1 Arrays Unidimensionais (Vetores) 96
7.1.1 Leitura e Escrita em arrays *98*
7.1.2 Inicialização de Vetores *99*
7.2 Matrizes (Arrays Bidimensionais) 100
7.2.1 Leitura de matrizes *102*
7.2.2 Escrita de uma matriz *103*
7.2.3 Como é a execução *104*
7.3 Exercícios Propostos 107

8 CADEIAS DE CARACTERES: STRINGS 111

8.1 Declaração duma string 111
8.2 Leitura de Strings 111
8.2.1 Leitura com fgets() 112
8.2.2 Leitura com scanf() 115
8.3 Funções da Biblioteca <string.h> 116
8.4 Extração Formatada com sscanf() 119
8.5 Tipo enumerado (enum) 121
8.6 Funções da Biblioteca <ctype.h> 121
8.7 Limpeza do buffer de entrada 124
8.8 Exercícios Propostos 125

9 FUNÇÕES 130

9.1 A Função main() 130
9.1.1 Argumentos de linha de comando 131
9.2 Por Que Usar Funções 134
9.3 Passagem de Parâmetros: Valor vs Referência 134
9.4 Estrutura Básica de uma Função 135
9.5 Protótipo de função 136
9.6 Tipos de Funções e Chamada 136
9.7 Escopo de Variáveis 139
9.7.1 Escopo Local 140
9.7.2 Escopo Global 140
9.7.3 Reutilização de Nomes de variáveis 141
9.8 Exercícios Propostos 142

10 APONTADORES 144

10.1 Declaração de Apontadores 145
10.2 Operações com apontadores 146
10.3 Apontadores e arrays uni e bidimensionais 148
10.3.1 Usar apontadores com matrizes 149
10.4 Alocação Dinâmica de Memória 151
10.4.1 malloc() 151
10.4.2 calloc() 152
10.4.3 realloc() 152
10.5 Vantagens e Riscos dos Apontadores 154
10.5.1 Vantagens 154
10.5.2 Riscos no Uso de Apontadores 154
10.6 Resumo das Operações com Apontadores 155
10.7 Exercícios Propostos 156

11 PASSAGEM DE PARÂMETROS POR REFERÊNCIA 158

11.1 COMO SE USA PASSAGEM DE PARÂMETRO POR REFERÊNCIA 158
11.2 DIFERENÇA EM RELAÇÃO À PASSAGEM DE PARÂMETRO POR VALOR 159
11.3 VETORES COMO PARÂMETROS 161
11.4 MATRIZES COMO PARÂMETROS 162
11.5 RETORNO DE VETORES 163
11.6 EXERCÍCIOS PROPOSTOS 165

12 STRUCTS: DEFININDO TIPOS COMPOSTOS 166

12.1 POR QUE USAR STRUCTS 166
12.2 DEFINIR E ACEDER A STRUCTS 167
12.3 SIMPLIFICAR COM TYPEDEF 170
12.4 VETORES DE STRUCTS 170
12.5 APONTADORES PARA STRUCT 172
12.6 STRUCTS ANINHADAS 174
12.7 EXERCÍCIOS PROPOSTOS 178

13 FICHEIROS EM C 182

13.1 FICHEIROS DE TEXTO 184
13.1.1 Abertura de Ficheiros 184
13.1.2 Escrita no Ficheiro Texto 186
13.1.3 Leitura de Ficheiro Texto 188
13.2 TESTES DE FIM DE FICHEIRO 191
13.2.1 Leitura com fscanf() 192
13.2.2 Leitura com fgets() 193
13.2.3 Leitura com fgetc() 193
13.3 FECHAMENTO DO FICHEIRO 194
13.4 FICHEIROS BINÁRIOS 195
13.4.1 Entrada e saída dos dados no ficheiro binário 196
13.4.2 Leitura e Escrita em Ficheiros Binários 197
13.4.3 Exemplo – Gravar e Ler uma struct: 198
13.5 GRAVAÇÃO E LEITURA DE UMA STRUCT 199
13.5.1 Ler uma struct de um ficheiro binário 200
13.5.2 Leitura até o fim de um ficheiro binário 201
13.6 VANTAGENS E DESVANTAGENS DOS FICHEIROS BINÁRIOS 203
13.7 EXEMPLOS DE LEITURA E ESCRITA EM FICHEIROS 204
13.7.1 Gravar dados num ficheiro de texto 204
13.7.2 Ler um vetor de estruturas de ficheiro binário 205
13.7.3 Função com gravação de dados (persistência) 205
13.7.4 Leitura de ficheiro texto com fgets() + sscanf() 206
13.8 MANIPULAÇÃO AVANÇADA DE FICHEIROS 206
13.8.1 fseek() 206

13.8.2 Exportação de Dados para Impressão ou Planilha *207*
13.9 EXERCÍCIOS PROPOSTOS 210

14 CRIAR BIBLIOTECA EM C 213

14.1 O QUE É UMA BIBLIOTECA? 213
14.2 ESTRUTURA DE UMA BIBLIOTECA 214
14.3 EXEMPLO COMPLETO DE BIBLIOTECA 214
14.4 COMPILAÇÃO DOS FICHEIROS 217
14.5 ORGANIZAÇÃO DAS BIBLIOTECAS 218
14.6 BIBLIOTECAS ÚTEIS EM C 219
14.6.1 math.h – Funções Matemáticas 219
14.6.2 string.h – Manipulação de Strings 220
14.6.3 ctype.h – Classificação de Caracteres 220
14.6.4 stdlib.h – Utilitários Diversos 221
14.6.5 time.h – Tempo e Data 222
14.7 BIBLIOTECAS PADRÃO 223

15 PROPOSTAS DE RESOLUCAO 225

15.1 CAPÍTULO 4 225
15.2 CAPÍTULO 5 230
15.3 CAPÍTULO 6 235
15.4 CAPÍTULO 7 243
15.5 CAPÍTULO 8 253
15.6 CAPÍTULO 9 262
15.7 CAPÍTULO 10 266
15.8 CAPÍTULO 11 271
15.9 CAPÍTULO 12 273
15.10 CAPÍTULO 13 288

16 REFERÊNCIAS BIBLIOGRÁFICAS 296

A LINGUAGEM C

Apesar de ter sido criada há mais de 50 anos, a linguagem C continua extremamente relevante no mundo da programação. Ao contrário do que muitas pessoas pensam, C não é uma linguagem "antiga e ultrapassada", muito pelo contrário, é a base sobre a qual muitas outras linguagens e sistemas foram construídos.

Hoje a linguagem C é amplamente utilizada em diversas áreas críticas da computação, tais como:

- Sistemas operativos, como Linux, Windows e MacOS, que têm grande parte do seu núcleo (*kernel*) escrito em C
- Sistemas embarcados, presentes em automóveis, eletrodomésticos, robôs, *routers* e equipamentos médicos, onde o controlo direto do hardware é essencial
- Programação de baixo nível, em que a proximidade com a máquina permite maior controlo sobre memória, desempenho e recursos do sistema
- Drivers, compiladores e bibliotecas de alto desempenho, onde a eficiência e o uso preciso dos recursos do sistema são fundamentais.

Aprender C é, portanto, uma forma de compreender como o computador realmente funciona por dentro. Além disso, é uma excelente base para outras linguagens como C++, Java ou mesmo Python, pois desenvolve o raciocínio lógico e o domínio sobre estruturas fundamentais da programação.

1 CONCEITOS BÁSICOS DE PROGRAMAÇÃO

1.1 Uma Breve História da Programação

Nos anos 1940, os primeiros computadores eram máquinas gigantescas que ocupavam salas inteiras e eram programados com fios e interruptores. Programadores como *Ada Lovelace* (considerada a primeira programadora da história) e *Alan Turing* (pai da computação teórica) pavimentaram o caminho para a programação moderna.

Hoje, programar é como dar vida a uma máquina—transformando ideias abstratas em soluções reais. Este livro foi criado para ser seu guia rápido, cheio de exemplos práticos e exercícios resolvidos, de forma a ajudá-lo a dominar a linguagem C de forma eficiente e divertida.

1.1.1 O Computador como Máquina de Processar Informação

Programar é instruir o computador para processar dados numa sequência lógica. Um computador é uma máquina incrivelmente rápida, capaz de realizar mil milhões de operações matemáticas e lógicas por segundo, entretanto faz apenas o que lhe mandam e da forma como se manda.

A linguagem C é uma linguagem compilada, o que significa que o código fonte escrito pelo programador, com a extensão `.c`, é convertido para um ficheiro executável, extensão `.exe`, numa linguagem de máquina. Este ficheiro executável é o que é corrido pelo computador.

Os passos aqui descritos não são executados directamente pelo compilador nem pelo computador de forma automática.

São pertencentes à lógica do programa, ou seja, são instruções que o programador escreve e que serão seguidas pelo programa compilado aquando da execução.

O código a seguir é um exemplo meramente ilustrativo, escrito na linguagem C e este trecho de código está a permitir calcular a média de três números.

Neste momento, não te preocupes com a sintaxe dos comandos, este é apenas um exemplo inicial. Com calma, estarás apto a escrever o teu próprio código.

O programa segue os passos:

1. Receber os três valores introduzidos pelo utilizador.
2. Somar os três valores.
3. Dividir o total por 3 para obter a média.
4. Exibir o resultado da média no ecrã

```c
int main() {
    float n1, n2, n3, media;
    // Passo 1: Receber os valores
    printf("Digite o primeiro numero: ");
    scanf("%f", &n1);
    printf("Digite o segundo numero: ");
    scanf("%f", &n2);
    printf("Digite o terceiro numero: ");
    scanf("%f", &n3);
    // Passo 2 e 3: Somar e dividir por 3
    media = (n1 + n2 + n3) / 3;
    // Passo 4: Exibir o resultado
    printf("A media e: %.2f\n", media);
    return 0;
}
```

Após a execução do programa, é apresentada a seguinte saída no ecrã, que corresponde ao resultado do cálculo realizado com base nos valores

fornecidos pelo utilizador. O programa pede três números, calcula a média aritmética desses valores e exibe o resultado formatado com duas casas decimais. Esta saída confirma que o programa seguiu correctamente os passos definidos no código, desde a leitura dos dados até à apresentação final do valor calculado.

```
Digite o primeiro numero: 10
Digite o segundo numero: 20
Digite o terceiro numero: 30
A media e: 20.00
```

1.2 Por que aprender a linguagem C

Programar consiste em conceber algoritmos, ou seja, conjuntos de passos lógicos para resolver um problema e traduzi-los para uma linguagem que o computador consiga interpretar. A linguagem C é uma das formas mais eficazes de o fazer.

Aprender a linguagem C continua a ser altamente recomendada para quem pretende adquirir uma base sólida em programação. Trata-se de uma linguagem estruturada, concisa e amplamente utilizada, cuja importância histórica e técnica permanece actual.

Muitos dos conceitos fundamentais das linguagens modernas, como Java, Python, C++ e C#, têm origem directa em C, pelo que dominar esta linguagem torna mais fácil a transição para outras além de permitir compreender melhor o seu funcionamento interno.

Para além, a linguagem C distingue-se por oferecer um controlo directo sobre o funcionamento do computador. Permite manipular a memória, interagir com portas e dispositivos físicos, e escrever software muito próximo do

hardware. Por essa razão, é amplamente utilizada em áreas como o desenvolvimento de sistemas operativos, *firmware*, *drivers*, microcontroladores e sistemas embarcados, incluindo projectos com Arduíno e outras plataformas de baixo custo.

Outra razão importante para aprender C é a sua eficiência. Os programas escritos nesta linguagem tendem a ser leves, rápidos e exigem poucos recursos do sistema. Esta característica torna C uma escolha natural para aplicações que requerem elevado desempenho, como jogos, simulações, sistemas em tempo real ou software industrial.

Aprender C exige disciplina e atenção aos detalhes, mas proporciona uma compreensão profunda da lógica de programação e do funcionamento interno das máquinas. É um excelente ponto de partida para qualquer pessoa que queira tornar-se um programador competente e tecnicamente sólido.

1.2.1 Representação de Algoritmos

Antes de escrever o código propriamente dito numa linguagem de programação como C, é importante planear a solução do problema. Esse planeamento pode ser feito de diferentes formas, conhecidas como formas de representação de algoritmos:

a) Narrativa (linguagem natural)

É uma descrição simples, clara e directa do que o algoritmo deve realizar, utilizando frases completas em português (ou noutra língua natural).

O objetivo desta forma de representação é transmitir, de maneira intuitiva, a lógica ou o fluxo de execução, sem recorrer a símbolos formais ou sintaxe de programação.

Por exemplo, pode-se descrever passo a passo as ações esperadas, de forma semelhante a instruções de um manual, permitindo que qualquer pessoa compreenda, mesmo sem conhecimentos técnicos, como por exemplo:

```
" Ler dois números fornecidos pelo utilizador,
calcular a sua soma e apresentar o resultado no
ecrã."
```

b) Pseudocódigo (une a linguagem humana e lógica de programação)

É uma descrição simples, clara e directa do que o algoritmo deve realizar, utilizando frases completas em português (ou noutra língua natural).

O objetivo é explicar a lógica ou o fluxo de execução de forma intuitiva, sem recorrer a símbolos formais ou sintaxe rígida de programação, permitindo que qualquer pessoa compreenda a sequência de passos, mesmo sem conhecimentos técnicos.

Pode-se ainda recorrer a uma escrita próxima da programação real, mas sem se preocupar com regras estritas de sintaxe.

Esta abordagem ajuda a organizar o raciocínio e a estruturar a resolução do problema antes da implementação do código propriamente dito.

Exemplo estruturado da narrativa apresentada na alínea a.

```
LEIA num1, num2
soma ← num1 + num2
ESCREVA soma
```

c) Fluxograma (representação gráfica)

Utiliza símbolos visuais (como setas, losangos, retângulos e outros elementos padronizados) para representar graficamente o fluxo de execução de um algoritmo.

Cada símbolo tem significado específico, como os retângulos que indicam ações ou processos, losangos as decisões e setas a ordem de execução.

O fluxograma auxilia na visualização do código e na compreensão de todas as etapas de forma clara e intuitiva, permitindo identificar facilmente o início, o fim, os pontos de decisão e as ações a executar.

A Figura 1-1 apresenta um exemplo simples de fluxograma, no qual se ilustram os passos de um algoritmo desde a entrada de dados até à apresentação do resultado.

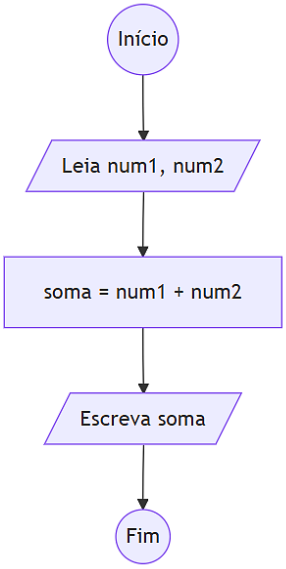


Figura 1-1 Exemplo de fluxograma

d) Linguagem de Programação (implementação real)

A linguagem de programação é como se escreve, de forma estruturada e precisa, os passos que o computador deve seguir para resolver um problema. No fundo, é a tradução do raciocínio lógico (algoritmo) para uma linguagem formal que o computador consegue interpretar. Neste exemplo, se utiliza a linguagem de programação C, uma das mais antigas e influentes.

A seguir, há um pequeno exemplo de código em C. Mesmo que ainda não compreendas cada comando, o mais importante neste momento é perceber que o programa está escrito como uma sequência de instruções muito claras, que o computador irá executar passo a passo.

Este programa pede ao utilizador para introduzir dois números, faz a soma desses dois valores e mostra o resultado no ecrã. As instruções `printf()` servem para apresentar mensagens ao utilizador, enquanto `scanf()` é usada para ler valores que o utilizador escreve no teclado. No exemplo a seguir, a variável `soma` guarda o resultado da adição.

```c
#include <stdio.h>

int main() {
    int num1, num2, soma;

    printf("Digite dois numeros: ");
    scanf("%d %d", &num1, &num2);

    soma = num1 + num2;
    printf("Soma: %d\n", soma);

    return 0;
}
```

Ao executar o programa, o computador apresenta a mensagem "`Digite dois numeros:`". Se o utilizador escrever os números `10` e `20`. O programa lê esses dois valores, soma-os e apresenta: "`Soma: 30`".

```
Digite dois numeros: 10 20
Soma: 30
```

Neste exemplo, o computador seguiu exactamente os passos definidos no código: mostrou uma mensagem, recebeu dois valores, somou-os e mostrou o resultado.

Este é um exemplo simples, mas já vos mostra como comunicar com o computador e obter respostas através de instruções escritas numa linguagem de programação.

1.3 Exercício Resolvido

Elaborar um pseudocódigo e o respetivo fluxograma para ler um valor real correspondente a uma temperatura em graus Celsius e calcular o valor equivalente em graus Fahrenheit.

Dica: $F = C * \frac{9}{5} + 32$

Em pseudocódigo, a resposta é a apresentada a seguir.

```
INÍCIO
    DECLARAR Celsius, Fahrenheit como reais
    LER Celsius
    Fahrenheit ← Celsius × (9/5) + 32
    ESCREVER Fahrenheit
FIM
```

Um fluxograma pode ser utilizado para representar a resolução deste problema de forma visual. A Figura 1-1Figura 1-2 apresentada a seguir, mostra essa representação passo a passo.

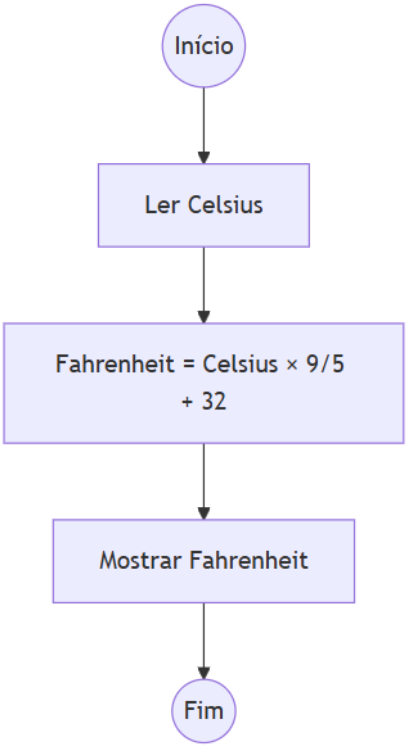


Figura 1-2 Converter graus Celsius para graus Fahrenheit

1.4 Pronto a Programar na Linguagem C?

Chegou o momento de trabalhar. Este livro foi concebido para que aprendas a programar praticando. A cada novo conceito, irás encontrar exemplos

simples, explicações acessíveis e pequenos desafios que te ajudarão a pensar como programador e a desenvolver as tuas próprias soluções.

A programação aprende-se com a experiência, com o erro e com a curiosidade. Não te preocupes se nem tudo fizer sentido à primeira, pois o mais importante é começares, experimentares e persistires.

Ao longo desta jornada, irás ganhar confiança e perceber que escrever código é, acima de tudo, uma forma de expressar ideias com lógica e clareza.

Então, é o momento de dar os primeiros passos e descobrir como funciona um programa em linguagem C.

1.4.1 Um programa C começa com uma estrutura mínima

Antes de avançarmos para os capítulos seguintes, é importante compreender que todos os programas escritos em C seguem uma estrutura básica. Essa estrutura inclui a função principal, chamada `main()`, onde o programa começa a ser executado, e a inclusão de bibliotecas, como `#include <stdio.h>` (abreviatura de *standard input/output*), que fornece funcionalidades essenciais, como a leitura e escrita no ecrã. Esta é a biblioteca principal da linguagem C.

A função `main()` é o ponto de partida do programa, e o seu conteúdo é delimitado por chavetas { }. No interior desse bloco, encontram-se as instruções que o computador irá executar, ou seja, o corpo do programa. O comando `return 0;`, colocado geralmente no final da função, indica ao sistema operativo que o programa terminou correctamente.

Na Figura 1-3, é apresentado um exemplo da estrutura mínima de um programa em C. Esta será a base sobre a qual se construirá os próximos exemplos, aos quais serão progressivamente acrescentadas variáveis, operações, decisões e repetições.

```c
#include <stdio.h>

int main() {
    //corpo do programa
     return 0;
}
```

Figura 1-3 Estrutura mínima de um programa em linguagem C.

1.4.2 Formatação livre e indentação

A linguagem C possui formatação livre, o que significa que os espaços, tabulações e quebras de linha não interferem no funcionamento do programa, desde que não apareçam dentro de identificadores, palavras-reservadas, números ou outros elementos que precisem estar juntos. Assim, um mesmo código pode ser escrito numa única linha ou distribuído em várias linhas e o compilador interpretará da mesma forma.

Apesar dessa liberdade, é boa prática manter um padrão de indentação, ou seja, um deslocamento visual do código para a direita, de forma a evidenciar a hierarquia e o agrupamento das instruções.

Um código bem indentado facilita a leitura, a manutenção e a deteção de erros lógicos, especialmente quando há estruturas de controlo aninhadas, como `if()`, `while()` e `for()`.

Resumidamente, as características importantes da formatação livre em C:

- Espaços e quebras de linha podem ser usados livremente para melhorar a legibilidade
- O compilador ignora a quantidade de espaços e tabulações extras
- A indentação não é obrigatória para o funcionamento, mas é fundamental como prática profissional
- Comentários (// ou /* ... */) podem ser colocados em qualquer ponto, desde que não interrompam palavras-chave ou constantes.
-

Para facilitar a perceção, apresentam-se a seguir dois códigos em C. O primeiro não possui indentação; o segundo executa os mesmos passos, mas com indentação adequada.

```c
#include <stdio.h>
int main(){
int i;
for(i=1;i<=5;i++){
if(i%2==0){
printf("%d é par\n",i);
}else{ printf("%d é ímpar\n",i); } }
return 0;
}
```

E a seguir se apresenta com uma boa indentação. É facilmente percetível que o código se torna bem mais legível e de fácil de compreensão.

```c
#include <stdio.h>

int main() {
    int i;

    for (i = 1; i <= 5; i++) {
        if (i % 2 == 0) {
            printf("%d é par\n", i);
        } else {
            printf("%d é ímpar\n", i);
        }
    }

    return 0;
}
```

2 *TIPOS DE DADOS E VARIÁVEIS*

2.1 Importância dos Tipos de Dados

Imagina que estás a organizar uma biblioteca. Para guardares os livros correctamente, precisas de prateleiras adequadas ao tamanho de cada um.

- Revistas ou folhetos finos cabem facilmente numa prateleira pequena, tal como o tipo de dados `char`, que serve para armazenar apenas um carácter.
- Romances ou livros de tamanho médio exigem uma prateleira um pouco maior, como o tipo `int`, utilizado para representar números inteiros.
- Enciclopédias volumosas, por sua vez, só cabem em prateleiras grandes — o que corresponde aos tipos `float` ou `double`, usados para números com casas decimais.

Na linguagem C, os tipos de dados funcionam como estas prateleiras: cada tipo é adequado para armazenar valores com determinados tamanhos e formatos.

Escolher o tipo errado pode causar problemas sérios, como perda de informação ou até *overflow*, que ocorre quando tentámos guardar um valor maior do que o tipo permite, tal como tentar enfiar uma enciclopédia numa gaveta demasiado pequena.

2.2 Tipos Primitivos de Dados

Na linguagem C, os tipos primitivos de dados (também designados por tipos básicos) constituem os blocos fundamentais utilizados para declarar variáveis e armazenar valores na memória.

Cada tipo define não só o género de informação que será guardada, como texto, número inteiro ou número com casas decimais, mas também a quantidade de memória que será reservada para esse valor, influenciando a forma como o computador interpreta e manipula os dados.

A Tabela 1 apresenta os principais tipos primitivos de dados disponíveis na linguagem C.

Tabela 1 - Tipos primitivos de dados em C

Tipo	Tamanho em bytes	Escala Numérica em bits	Descrição
`char`	1	8 bits (-128 a 127 ou 0 a 255)	Armazena um único carácter (código ASCII)
`int`	4	32 bits (-2 147 483 648 a 2 147 483 647)	Armazena números inteiros
`float`	4	32 bits ($\pm 3.4 \times 10^{-38}$ a $\pm 3.4 \times 10^{38}$)	Para números reais com precisão simples
`double`	8	64 bits ($\pm 1.7 \times 10^{-308}$ a $\pm 1.7 \times 10^{308}$)	Para números reais com precisão dupla
`void`	0	Sem valor	Indica ausência de tipo

Observações:

- Estes são os tamanhos padrão em sistemas operativos modernos (Windows, Linux, macOS) com arquitectura de 32 ou 64 bits.

– Os valores para tipos de ponto flutuante (`float` e `double`) correspondem ao padrão IEEE 754, amplamente adoptado pelas linguagens de programação.

2.2.1 Atribuição, Inicialização e Inicialização Composta

O símbolo = em C corresponde ao operador de atribuição, e se utiliza para guardar um valor numa variável. A operação realiza-se da direita para a esquerda: o valor que à direita é avaliado e depois se coloca na variável que está à esquerda.

É importante perceber que o símbolo = não significa "igual" como na matemática. Em programação, é um operador de atribuição e serve para guardar um valor dentro de uma variável.

Pode-se interpretar o operador de atribuição como: “recebe o valor de” ou “fica com o valor de”.

2.2.2 Atribuição simples

A atribuição simples ocorre quando uma variável recebe um valor após ser declarada.

```
int x;
x = 10;
```

Neste exemplo, se declara a variável `x` na primeira linha e, a seguir, recebe o valor `10`.

2.2.3 Inicialização

A inicialização consiste em declarar a variável e atribuir-lhe de imediato um valor.

```
int x = 10;
```

Neste caso, a variável `x` já armazena o valor `10` logo no momento da criação desta variável. Esta forma é mais compacta e evita esquecer a atribuição posterior.

2.2.4 Inicialização composta

A inicialização composta permite atribuir vários valores a um conjunto de variáveis de forma simultânea. É usada, por exemplo, com arrays ou estruturas.

Um exemplo com array, que será explorado no momento adequado é:

```
int numeros[4] = {5, 10, 15, 20};
```

Neste caso, o array `numeros` é criado com quatro posições e cada uma recebe um valor inicial.

Exemplo com estrutura:

```
struct Pessoa {
    int idade;
    char sexo;
};
struct Pessoa p = {30, 'M'};
```

Neste trecho de código, a variável `p` do tipo `Pessoa` se inicia com dois valores: `30` para a variável `idade` e `'M'` para o `sexo`.

2.2.5 Atribuições encadeadas

É possível efectuar várias atribuições na mesma linha.

A expressão `i = j = 10;` é válida em C. De início, se atribui o valor `10` para a variável `j`. De seguir, o valor actual de `j` é então atribuído à variável `i`. Ao fim, ambas as variáveis ficam com o valor `10`.

```
int i, j;
i = j = 10;
```

Embora esta forma seja permitida, pode dificultar a leitura do código em situações mais complexas. Em contextos didácticos, recomenda-se usar uma atribuição por linha para garantir maior clareza.

2.2.6 Exemplo ilustrativo

Na linguagem C, como já visto anteriormente, os tipos primitivos permitem armazenar diferentes tipos de informação de forma eficiente. O exemplo a seguir mostra como declarar variáveis com diferentes tipos — `char, int, float e double` — e como apresentar os respectivos valores no ecrã com a função `printf()`.

```c
#include <stdio.h>

int main() {
    char simbolo = '@';
    int numero = 2140000000;
    float temperatura = 36.1;
    double pi = 3.1415926535;

    printf("Simbolo: %c\n", simbolo);
    printf("Numero inteiro: %d\n", numero);
    printf("Temperatura: %.1fC\n", temperatura);
    printf("PI: %.10lf\n", pi);

    return 0;
}
```

Neste exemplo a variável `simbolo` armazena um único carácter (`@`), do tipo `char`;. Já a variável `numero` guarda um número inteiro, dentro do intervalo permitido para o tipo `int`. A variável `temperatura` é do tipo `float` e representa um valor com uma casa decimal. E por fim, a variável `pi` é declarada do tipo `double` e possui maior precisão com números decimais.

Para além, a variável `símbolo` recebe o valor `@`. Este valor está entre aspas simples (ou *plics*), porque se trata de um caractere, do tipo `char`.

Seguindo a mesma lógica, outras variáveis também recebem valores, tal como no caso da variável `temperatura`, que recebe o valor `36.1`.

Os comandos `printf()` utilizam especificadores de formato para apresentar os dados correctamente: `%c` para `char`, `%d` para `int`, `%.1f` para `float`, com uma casa decimal, `%.10lf` para `double`, a mostrar 10 casas decimais.

Este pequeno programa mostra na prática, como os tipos de dados influenciam a forma como a informação se armazena num programa e se apresenta. Com o uso adequado dos tipos primitivos, é possível representar dados com a precisão necessária, a respeitar os limites de cada tipo.

2.3 Modificadores de Tipo

Os modificadores de tipo em C são utilizados para ajustar o tamanho e/ou o intervalo de valores que um tipo primitivo pode representar. Estes modificadores oferecem ao programador mais controlo sobre o uso da memória e sobre os limites dos dados armazenados, permitindo optimizar o programa conforme as necessidades específicas de cada situação.

Os principais modificadores de tipo são os seguintes:

- `signed` (com sinal): Permite armazenar valores positivos e negativos. É o comportamento padrão para os tipos inteiros.
- `unsigned` (sem sinal): Permite apenas valores positivos, aumentando significativamente o limite superior, uma vez que não é necessário reservar espaço para representar valores negativos.
- `short`: Reduz o espaço ocupado na memória. Ideal quando se deseja economizar espaço e os valores são pequenos.
- `long`: Aumenta o espaço de armazenamento, permitindo representar valores inteiros maiores.

A Tabela 2 apresenta uma versão ampliada, com tipos adicionais frequentemente utilizados em C, incluindo informações relevantes sobre o intervalo numérico e a sua utilização típica.

Tabela 2 Tipos de dados e modificadores em C

Tipo	Tamanho em bytes	Escala Numérica em bits	Observações
`char`	1	8 bits (-128 a 127 ou 0 a 255)	Armazena caracteres ASCII
`short int`	2	16 bits (-32.768 a 32.767)	Inteiro curto, pouca memória
`unsigned short int`	2	16 bits (0 a 65.535)	Apenas valores positivos curtos
`int`	4	32 bits (-2.147.483.648 a 2.147.483.647)	Inteiro padrão, mais usado atualmente
`unsigned int`	4	32 bits (0 a 4.294.967.295)	Inteiro sem sinal, usado para contadores
`long int`	4 ou 8	32 ou 64 bits (-2.147.483.648 a 2.147.483.647 ou ±9.223.372.036.854.775.808)	Inteiro longo, depende do sistema
`unsigned long int`	4 ou 8	32 ou 64 bits (0 a 4.294.967.295 ou 0 a 18.446.744.073.709.551.615)	Sem sinal, depende da arquitetura
`long long int`	8	64 bits (-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807)	Garantidamente inteiro muito longo
`float`	4	32 bits (3.4E-38 a 3.4E+38)	Precisão simples, padrão IEEE 754
`double`	8	64 bits (1.7E-308 a 1.7E+308)	Precisão dupla, mais usado para cálculos científicos

Tipo	Tamanho em bytes	Escala Numérica em bits	Observações
`long double`	8, 12 ou 16	≥64 bits (aprox. 1.2E-4932 a 1.2E+4932)	Alta precisão, depende da arquitetura
`void`	0	sem valor	Ausência de valor

Notas adicionais importantes:

- **Portabilidade**: O tamanho de tipos como `long int` e `long double` pode variar conforme o compilador ou sistema operativo.
- ***Unsigned*** (sem sinal): Tipos sem sinal (`unsigned`) são usados quando o valor negativo não é necessário e se quer aumentar o limite positivo disponível.
- **Tipos especiais**: Recomenda-se o uso de tipos definidos na biblioteca `<stdint.h>` como `int32_t`, `uint32_t`, `int64_t`, para garantir precisão e portabilidade especialmente em aplicações críticas.

2.4 Declaração de Variáveis

Uma variável é uma unidade de armazenamento de dados que representa um espaço reservado na memória do computador, identificado por um nome único. Esse espaço pode conter valores que são atribuídos, modificados e utilizados durante a execução de um programa, conforme as instruções definidas pelo programador.

Sintaxe:

```
tipo nome_da_variavel;
```

Exemplo:

```
int idade;           // variável inteira
float peso;          // variável com casas decimais
char inicial_nome;   // variável para um único caractere
```

2.4.1 Regras básicas para nomeação de variáveis em C:

a) Primeiro carácter deve ser uma letra (`a-z, A-Z`) ou um sublinhado (_). Exemplo:

Nomes válidos: `idade, _idade`

Nomes inválidos: `1idade, #idade`

b) Os nomes das variáveis são *case-sensitive*, ou seja, distinguem entre maiúsculas de minúsculas. Exemplo:

```
int Idade, idade; // Duas variáveis diferentes
```

c) O nome pode conter letras, números e sublinhado após o primeiro carácter. Exemplo:

```
float valor1, nota_final, _resultado;
```

d) Não são permitidos espaços nem caracteres especiais (exceto o *underline*). Exemplo:

Nomes válidos: `preco_produto, _preco`

Nomes inválidos: `preço produto, preço-produto`

Para além das regras apresentadas, não se pode utilizar palavras-chave ou palavras reservadas.

Na linguagem C existem 32 palavras reservadas que fazem parte da própria sintaxe da linguagem. Essas palavras têm significados especiais e são usadas

para definir estruturas de controlo, tipos de dados, operadores, dentre outros elementos essenciais do programa. Por esse motivo, não é permitido utilizá-las como nomes de variáveis, funções ou identificadores. Se tentar usar uma dessas palavras como nome de variável, o compilador vai gerar erro, pois entenderá que estás a tentar usar uma palavra com função específica fora do seu contexto correto. As palavras reservadas são apresentadas na Tabela 3.

Tabela 3 Palavras reservadas na linguagem C

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

Ao escrever programas, é importante adoptar boas práticas na escolha dos nomes das variáveis. Sempre que possível, os nomes devem ser claros e descritivos, de forma a indicar a função ou o tipo de dado que representam. Por exemplo, utilizar nomes como `altura` ou `contadorNotas` torna o código mais legível e fácil de compreender.

Além disso, recomenda-se que se mantenha um padrão consistente de escrita ao longo de todo o programa, seja utilizando *camelCase* (como `numeroTotal`), *snake_case* (como `numero_total`) ou *PascalCase* (como `NumeroTotal`). A consistência ajuda a evitar erros e facilita a leitura do código, tanto por quem o escreve, como por outros que o venham a ler ou manter.

A seguir é apresentado um exemplo simples de declaração de variáveis:

```
// Declaração das variáveis
int idadeAluno;
float alturaAluno;
char grupoSanguineo, letra;
```

2.5 Inicialização de Variáveis

Após declarar uma variável, é importante atribuir-lhe um valor inicial, ou seja, inicializá-la. Isto evita comportamentos inesperados no programa e permite que a variável comece a ser utilizada com um valor definido. A inicialização pode ser feita directamente no momento da declaração ou numa instrução posterior.

A forma recomendada consiste em declarar e inicializar a variável numa única linha. Este método reduz o risco de esquecer a atribuição de um valor e torna o código mais compacto e fácil de compreender, como se vê no exemplo a seguir:

```
// Declara e inicia as variáveis
int idadeAluno = 20;
float alturaAluno = 1.78;
char grupoSanguineo = 'O', letra = 'z';
```

Como alternativa, é possível separar a declaração da inicialização. Embora também seja válida, esta abordagem pode aumentar o risco de utilização de variáveis ainda não inicializadas, especialmente em programas maiores ou mais complexos.

```
// Declara as variáveis
int idadeAluno;
float alturaAluno;
char grupoSanguineo, letra;
// Inicia as variáveis
idadeAluno = 20;
alturaAluno = 1.78;
grupoSanguineo = 'O';
letra = 'z';
```

Atenção: Variáveis declaradas sem inicialização contêm valores indefinidos, ou seja, lixo de memória. Isso pode provocar comportamentos imprevisíveis no programa, como resultados errados ou falhas de execução. Por essa razão, é altamente recomendável inicializar sempre as variáveis antes de as utilizar.

Exemplo incorreto:

```
int valor;
printf("%d", valor);
// valor indefinido, resultado imprevisível!
```

Exemplo correcto:

```
int valor = 0;
printf("%d", valor);
// valor definido claramente como 0
```

2.6 Constantes

As constantes são valores fixos que não podem ser alterados durante a execução do programa. Utilizadas para representar dados que devem

permanecer inalteráveis, como o valor de *pi*, a quantidade de dias num ano ou mensagens fixas no ecrã. Em C, existem duas formas principais de definir constantes:

a) Usar const (constante em tempo de compilação):

Utiliza-se a palavra reservada `const` seguida do tipo do dado, para indicar que o valor atribuído não poderá ser alterado durante a execução do programa.

```
const int DIAS_ANO = 365;
const float GRAVIDADE = 9.8;
```

Ao definir uma constante, tenha em mente que é, sempre, obrigatório indicar o tipo da constante.

O uso de constante é um tipo de abordagem que oferece maior segurança, pois o compilador impede alterações indevidas ao valor definido.

Para além, as constantes declaradas com `const` respeitam as regras de escopo e podem ser utilizadas em contextos mais seguros e estruturados.

b) Usar #define (pré-processador):

A diretiva `#define` permite criar constantes através de uma substituição textual directa feita antes da compilação do programa, durante a fase de pré-processamento. Como exemplo de uso, tem-se:

```
#define PI 3.14159
#define NOME_PROGRAMA "Calculadora"
```

Tenhas atenção, que não se utiliza ponto e vírgula (;) no final da linha, pois o compilador percebe que deve substituir o texto por toda a expressão, a incluir o ponto e vírgula.

Por convenção, os nomes das constantes definidas com `#define` são escritos em letras maiúsculas, com palavras separadas por sublinhados (_), para facilitar a leitura e distingui-las claramente das variáveis.

Atenção:

- Não é obrigatório usar letras maiúsculas em `const`, mas podes fazê-lo por clareza ou para uniformizar com o estilo de `#define`.
- O mais importante é ser coerente ao longo do código: se usares maiúsculas para constantes, mantém esse padrão sempre.

No tópico 2.8, deste capítulo, há uma explicação mais detalhada do uso da diretiva `#define`, entretanto, o código seguinte mostra como definir e utilizar constantes na linguagem C, usando tanto a diretiva `#define` como a palavra reservada `const`. De seguir, se exibe as constantes no ecrã com a função `printf()`.

```c
#include <stdio.h>
#define PI 3.14159
const int DIAS_SEMANA = 7;

int main() {
    printf("Valor de PI: %.5f\n", PI);
    printf("Dias numa semana: %d\n", DIAS_SEMANA);
    return 0;
}
```

O programa tem a seguinte saída no ecrã:

```
Valor de PI: 3.14159
Dias numa semana: 7
```

2.7 Como trocar o conteúdo de duas variáveis

Um problema muito comum, e recorrente em programas escritos em qualquer linguagem de programação, é a troca de valores entre duas variáveis.

Embora pareça simples à primeira vista, essa operação exige atenção para que os valores não se percam durante o processo.

Entretanto, não basta escrever A = B seguido de B = A, pois isso faz com que o valor original de A se perca. Veja porquê:

```
int A, B;
A = 10;
B = 20;
// Tentativa incorreta:
A = B;  // A está a guardar 20
B = A;  // Portanto B também vale 20
// O valor original de A (10) perdeu-se
```

Para que a troca funcione corretamente, é obrigatório, em C, uma variável auxiliar, como neste exemplo:

```
int A, B, auxiliar;
A = 10;
B = 20;
auxiliar = A;  // guarda o valor original de A
A = B;     // A recebe o valor de B
B = auxiliar;  // B recebe o valor original de A
```

Após essas três instruções, se garante que A está a guardar o valor 20 e B está a guardar o valor 10, ou seja, os valores foram trocados com sucesso.

2.8 Constantes simbólicas e diretivas de pré-processador

A diretiva `#define` não cria uma variável, mas sim uma substituição textual feita pelo pré-processador antes da compilação e, como visto, o uso desta diretiva é para associar um nome simbólico a um valor constante, para facilitar a leitura e manutenção do código e como exemplo de uso, tem-se:

```
#define PI 3.14159
#define TAMANHO 50
```

A partir dessas definições, o compilador substitui todas as ocorrências de `PI` por `3.14159`, e `TAMANHO` por `50`, como se esses valores tivessem sido escritos diretamente no código.

Assim, pode-se escrever:

```
float area = PI * raio * raio;
int vetor[TAMANHO];
```

Essa abordagem torna o código mais claro e facilita futuras alterações. Sempre que for necessário alterar um valor como, por exemplo, o número 50, basta modificar a definição de `TAMANHO` no início do programa.

Além de valores numéricos, o `#define` pode ser usado para definir macros simples, ou seja, trechos curtos de código que se comportam como funções *inline*.

A seguir é apresentado um exemplo de definição de macro.

```
#define DOBRO(x) ((x) * 2)
```

Nesse caso, toda vez que se escreve o comando `DOBRO(5)` no código fonte, o compilador substituirá por ((`5`) * `2`).

Mas cuidado, pois como `#define` é apenas uma substituição textual, não há verificação de tipos, escopo ou limites de uso. Por isso, parênteses devem ser usados com atenção, especialmente em macros com expressões.
Nos próximos capítulos, serão apresentados exemplos práticos de utilização da diretiva `#define`, como na definição do tamanho de vetores ou na configuração de valores fixos para controlo de fluxo. Sempre que necessário, cada novo uso será explicado com mais detalhe.

2.9 Exercícios Propostos

1. Definição de constantes: Definir uma constante chamada TAXA_JUROS com valor `0.07`, utilizando as duas formas possíveis na linguagem C. Exibir no ecrã.
Explicar a diferença entre elas e indique qual oferece maior segurança durante a compilação.

2. Declaração e inicialização de variáveis: Declarar quatro variáveis com os seguintes tipos e valores iniciais:

- Um carácter com o símbolo @
- Um número inteiro com valor 1000
- Um número decimal com valor `3.14`
- Um número decimal de maior precisão com valor `9.81`

3. Identificação do tipo apropriado: Analisar os valores e, para cada um, indica o tipo de dado mais apropriado em C, justificando a escolha:

1. Número de cadeiras numa sala d'aula
2. Temperatura diária
3. Letra inicial do nome
4. População mundial

3 OPERADORES

Os operadores são símbolos especiais que realizam operações específicas sobre variáveis ou valores, permitindo criar expressões para cálculos, comparações e operações lógicas dentro dos programas em C.

3.1 Operadores Aritméticos

Realizam cálculos matemáticos básicos entre variáveis ou valores numéricos, conforme apresentados na Tabela 4

Tabela 4 - Operadores aritméticos em C

Operador	Descrição	Exemplo	Resultado (para x=5)
+	Soma	x+3	8
-	Subtração	x-2	3
*	Multiplicação	x*4	20
/	Divisão	x/2	2 (inteira) 2.5 (float)
%	Módulo (resto da divisão entre dois valores inteiros)	x%2	1

Atenção

a) Divisão entre dois números inteiros

Na linguagem C, ao dividir dois valores inteiros, o resultado também será um número inteiro, porque a parte decimal é truncada, ou seja, é descartada sem arredondamento. Por exemplo:

```
int res = 5/2; // Resulta em 2 e não 2.5
```

b) Conversão de tipo (*casting*):

Para obter um resultado com casas decimais numa divisão entre inteiros, é necessário converter explicitamente um dos operandos para `float`. Isso é feito com o operador de *cast*, como neste exemplo:

```
int a = 5;
int b = 2;
float res = (float)a/b; // Resultado: 2.5
```

Ao fazer `(float)a`, o valor de a passa a ser tratado como número decimal (ponto flutuante) e o resultado da divisão é também um número decimal.

c) Operador de resto (módulo):

O operador % (módulo) é usado para obter o resto da divisão entre dois números inteiros. Por exemplo:

```
int resto1 = 5%2;  // Resultado: 1
int resto2 = 15%3; // Resultado: 0
```

O operador % só pode ser usado com valores inteiros. É especialmente útil em situações como verificar se um número é par ou ímpar (x % 2 == 0) ou para resolver problemas de divisibilidade. Mas há muitas mais aplicações.

3.2 Operadores Relacionais

Os operadores relacionais são utilizados para comparar dois valores ou variáveis. O resultado dessas comparações é sempre um valor inteiro lógico, representado por:

- 0 → falso (condição falsa, ou seja, não satisfeita)

- 1 → verdadeiro (condição verdadeira, ou seja, satisfeita)

Estes operadores são fundamentais em estruturas condicionais e ciclos, permitindo ao programa tomar decisões com base nas condições estabelecidas.

Observação importante:

Na linguagem C, o valor 0 é interpretado como falso, e qualquer valor diferente de 0 é interpretado como verdadeiro nas estruturas de controlo.

No entanto, os operadores relacionais são implementados para devolver sempre 1 (e não qualquer outro valor diferente de zero) quando a condição é verdadeira. Esse comportamento assegura consistência lógica ao comparar expressões.

A Tabela 5 apresenta os operadores relacionais, com uma breve descrição e um exemplo de uso e resultado para cada operador.

Tabela 5 Operadores relacionais em C

Operador	Descrição	Exemplo	Resultado
==	Igualdade	`5 == 5`	`1`
!=	Diferença	`5 != 3`	`1`
<	Menor que	`13 < 4`	`0`
<=	Menor ou igual a	`4 <= 4`	`1`
>	Maior que	`8 > 9`	`0`
>=	Maior ou igual a	`7 >= 5`	`1`

A seguir um exemplo que utiliza os operadores relacionais.

```c
int idade = 20;
printf("%d", idade >= 18);
// Imprime 1 (verdadeiro)
```

3.3 Operadores Lógicos

Os operadores lógicos são utilizados para conectar duas ou mais condições relacionais, que permitem formar condições compostas mais complexas. São especialmente úteis em situações onde é necessário verificar múltiplas condições simultaneamente.

Os três principais operadores lógicos em C são:

- `&&` **(E lógico):** Retorna verdadeiro (1) somente se todas as condições forem verdadeiras.
- `||` **(OU lógico):** Retorna verdadeiro (1) se pelo menos uma das condições for verdadeira.
- `!` **(NÃO lógico):** Inverte o valor lógico da condição avaliada (transforma verdadeiro em falso e vice-versa).

Esses operadores permitem criar verificações complexas em poucas linhas, simplificando a lógica dos programas.

A Tabela 6 apresenta um resumo destes operadores relacionais em C.

Tabela 6 Operadores relacionais em C

Operador	Nome	Exemplo	Explicação
`&&`	E lógico	`(x > 0 && x < 10)`	Verdadeiro apenas se ambas as condições forem verdadeiras.
`||`	OU lógico	`(x < 0 || x > 10)`	Verdadeiro se pelo menos uma das condições for verdadeira.
!	NÃO lógico	`!(x == 0)`	Inverte o resultado (falso vira verdadeiro e vice-versa).

Considere este exemplo que combina o uso de operadores relacionais e lógicos num contexto. Para verificar se uma pessoa pode conduzir. Para isso, é necessário que tenha pelo menos 18 anos e possua carta de condução.

```c
int main() {
    int idade = 20;
    int temCarteira = 1;
    int podeDirigir = (idade >= 18) && temCarteira;

    printf("Pode dirigir? %d\n", podeDirigir);
    return 0;
}
```

Neste exemplo:

- A variável `idade` guarda o valor 20 e a variável `temCarteira` vale 1 (verdadeiro).
- A expressão `(idade >= 18) && temCarteira` será avaliada. Se for verdadeira, o resultado será 1; caso contrário, será 0.
- O resultado é guardado na variável `podeDirigir` e depois apresentado com `printf()`.

```
Pode dirigir? 1
```

Este é um simples exemplo, apenas ilustrativo, que permite trabalhar com operadores relacionais e lógicos, sem introduzir ainda estruturas de controlo, como o `if()`.

3.4 Operadores de Atribuição Composta

Os operadores de atribuição composta permitem simplificar operações em que uma variável é atualizada com base em seu próprio valor. Em vez de escrever a operação completa (por exemplo, `x = x + 5`), pode-se utilizar uma forma abreviada (`x += 5`).

Estes operadores tornam o código mais compacto e fácil de ler, especialmente em situações muito comuns, como no uso de contadores e acumuladores dentro de ciclos de repetição.

Antes de apresentar esses operadores, convém compreender dois conceitos fundamentais na programação:

- **Contador**: é uma variável utilizada para contar quantas vezes algo ocorre. Normalmente, o contador começa em zero e é incrementado a cada ocorrência. Exemplo: `contador = contador + 1;`
- **Acumulador**: é uma variável utilizada para somar (ou acumular) sucessivos valores durante a execução do programa, guardando o total resultante ao longo do tempo. Exemplo: `soma = soma + valor;`

Estes conceitos serão úteis nos capítulos seguintes, quando abordarmos ciclos `for, while` e outras estruturas que repetem operações. Por agora, interessa perceber que os operadores de atribuição composta são formas práticas de actualizar o valor de uma variável.

Voltando aos nossos **operadores de atribuição composta**, estes são comandos que combinam a operação aritmética ou lógica com a atribuição de valor a uma variável, tornando o código mais conciso e legível.

Na prática, permitem executar uma operação e armazenar o resultado na própria variável de forma direta, evitando escrever a variável duas vezes.

Estes são os comandos utilizados nesta definição:

- **`+=`** → soma e atribui à variável a esquerda do operador
- **`-=`** → subtrai e atribui à variável a esquerda do operador
- **`*=`** → multiplica e atribui à variável a esquerda do operador
- **`/=`** → divide e atribui à variável a esquerda do operador
- **`%=`** → calcula o resto entre dois inteiros e atribui à variável a esquerda do operador

Utilizar operadores de atribuição composta é considerado uma boa prática de programação. A Tabela 7 apresenta, através de exemplos, uma comparação de uso e uma breve explicação.

Tabela 7 Exemplos de uso dos operadores de atribuição composta

Forma longa	Forma curta	Explicação
`x = x + 5;`	`x += 5;`	Adiciona 5 a x
`y = y - 2;`	`y -= 2;`	Subtrai 2 de y
`z = z * k;`	`z *= k;`	Multiplica z por k
`w = w / 4;`	`w /= 4;`	Divide w por 4
`n = n % 2;`	`n %= 2;`	Resto da divisão por 2

3.5 Operadores de Incremento e Decremento

Os operadores de incremento (++) e decremento (--) são utilizados para alterar o valor de uma variável em exatamente **uma** unidade. São amplamente usados para contadores ou em laços de repetição.

Esses operadores possuem duas formas de uso:

- **Pré-incremento (++x) e Pré-decremento (--x)**: O valor da variável é alterado de 1 unidade antes de ser utilizado numa expressão.
- **Pós-incremento (x++) e Pós-decremento (x--)**: O valor original da variável é utilizado na expressão primeiro e somente depois é feita a alteração em 1 unidade.

Esses operadores simplificam a escrita do código, tornando-o mais compacto e fácil de entender especialmente em *loops* ou iterações frequentes. Contudo, é importante estar atento à diferença entre **pré** e **pós**, pois afetam diretamente o resultado da expressão utilizada.

A Tabela 8 apresenta uma breve explicação destes comandos.

Tabela 8 - Breve explicação do pré e pós incremento/decremento

Operador	Descrição	Explicação
`++x`	Pré-incremento	Incrementa antes do uso
`x++`	Pós-incremento	Incrementa depois do uso
`--x`	Pré-decremento	Decrementa antes do uso
`x--`	Pós-decremento	Decrementa depois do uso

Já a Tabela 9 exemplifica passo a passo o comportamento dos operadores de incremento (++) e decremento (--)

O exemplo considera sempre que a = 10 no início de cada linha, permitindo observar passo a passo como o valor da variável é alterado antes ou depois de ser utilizado na expressão.

Assim, é possível compreender a diferença entre executar `++a` (incremento antes de usar) e `a++` (incremento depois de usar), bem como `--a` e `a--` no caso do decremento.

Tabela 9 Exemplo dos comandos de pré e pós incremento/decremento

Instrução	Explicação	Valor final de a	Valor final de b
`b = a++;`	b recebe o valor **actual** de a (10), **depois** a é incrementado.	`11`	`10`
`b = a--;`	b recebe o valor **actual** de a (10), **depois** a é decrementado.	`9`	`10`
`b = ++a;`	a é **incrementado primeiro** (a passa a 11), depois b recebe o valor.	`11`	`11`
`b = --a;`	a é **decrementado primeiro** (a passa a 9), depois b recebe o valor.	`9`	`9`

3.6 Precedência de Operadores

A **precedência de operadores** define a ordem em que as operações são executadas dentro de uma expressão.

Operadores com maior prioridade são avaliados primeiro, enquanto os de menor prioridade são resolvidos depois.

Compreender bem esta ordem é fundamental para garantir que as expressões produzem os resultados esperados, evitando interpretações incorretas do compilador.

Em C, a ordem de prioridade dos operadores, da maior para a menor, está apresentada na Tabela 10**.** Esta tabela é uma referência rápida para programadores, ajudando a interpretar expressões complexas e a decidir se é necessário o uso de parênteses para alterar a ordem padrão de execução.

Tabela 10 Ordem de precedência dos operadores em C

Prioridade	Operadores	Descrição
1 (mais alta)	( )	Parênteses para agrupamento
2	!, ++, --	Operadores unários (lógicos e incremento/decremento)
3	*, /, %	Operadores aritméticos multiplicativos
4	+, -	Operadores aritméticos aditivos
5	<, <=, >, >=	Operadores relacionais
6	==, !=	Operadores de igualdade/diferença
7	&&	Operador lógico E
8	||	Operador lógico OU
9 (mais baixa)	=, +=, -=, *= ...	Operadores de atribuição

Dica prática: Para evitar dúvidas ou possíveis erros, utilize sempre parênteses para deixar explícito qual operação deve ser executada primeiro. Além de prevenir interpretações incorretas pelo compilador, esta prática melhora significativamente a legibilidade do código, facilitando a leitura por outros programadores (e até por si mesmo no futuro).

Veja o que ocorre no exemplo a seguir.

```
int resultado = (5 + 3) * 2;
// Resultado é 16
int errado = 5 + 3 * 2;
// Resultado é 11 (sem parênteses)
```

No primeiro caso, o uso dos parênteses força que a soma seja realizada antes da multiplicação, garantindo que o resultado obtido seja exatamente o pretendido.

Sem os parênteses, a expressão seguiria a ordem de precedência padrão, podendo gerar um resultado diferente do esperado.

Exercícios resolvidos com explicação detalhada por linha de código:

Apresenta-se exemplos de código acompanhados de uma análise de cada instrução.

O objetivo é mostrar não apenas o que o código faz, mas também como e por que cada linha é escrita dessa forma, para se compreender a lógica passo a passo e consolidar o raciocínio de programação.

a) O que será impresso ao fim deste trecho de código?

```
int x = 10;
int y = x-- + 5;
int z = ++x + 2;
int a = y - z;
a = -a;
x = x + a;
y = y + x - z;
z = z * a;
printf("x = %d\n", x);
printf("y = %d\n", y);
printf("z = %d\n", z);
printf("a = %d\n", a);
```

Explicação linha a linha:

1. `x = 10` → inicia x com 10
2. `y = x-- + 5`
   - Usa `x = 10` na expressão
   - `y = 10 + 5 → y = 15`
   - Depois, x é decrementado: `x = 9`
3. `z = ++x + 2`
   - `++x` incrementa `x` antes de usar → `x = 10`
   - `z = 10 + 2 → z = 12`
4. `a = y - z`
   - `a = 15 - 12 → a = 3`
5. `a = -a`
   - `a = -3`
6. `x = x + a`
   - `x = 10 + (-3) → x = 7`
7. `y = y + x - z`
   - `y = 15 + 7 - 12 → y = 10`
8. `z = z * a`
   - `z = 12 * (-3) → z = -36`

Resposta final:

```
x = 7
y = 10
z = -36
a = -3
```

b) O que será impresso ao fim deste trecho de código?

```c
int a = 4;
int b = 7;
int c = a++ + b;     // linha A
int d = --b + c;     // linha B
c = -c;              // linha C
a = a + b - d;       // linha D
b = c + d - a;       // linha E
d = a * b;           // linha F
printf("a = %d\n", a);
printf("b = %d\n", b);
printf("c = %d\n", c);
printf("d = %d\n", d);
```

Explicação linha a linha:

a) `a = 4, b = 7` → inicializações

b) `c = a++ + b`

- Usa `a = 4` antes de incrementar
- `c = 4 + 7 = 11`
- Depois, `a = 5`

c) `d = --b + c`

- Primeiro decrementa b: `b = 6`
- `d = 6 + 11 = 17`

d) `c = -c`

- `c = -11`

e) `a = a + b - d`

- `a = 5 + 6 - 17 = -6`

f) `b = c + d - a`

- `b = (-11) + 17 - (-6) = 12`

g) `d = a * b`

- `d = -6 * 12 = -72`

4 ENTRADA E SAÍDA DE DADOS

4.1 Biblioteca Padrão stdio.h

Em C, uma biblioteca é um conjunto de funções e definições previamente implementadas, que podem ser reutilizadas nos programas. Estas bibliotecas evitam que se reescreva código para tarefas comuns, como ler dados do utilizador, apresentar informações no ecrã ou realizar cálculos matemáticos.
A biblioteca `stdio.h` (*Standard Input/Output)* é uma das mais importantes da linguagem C. Nesta biblioteca estão as funções de entrada e saída, como `printf`() para escrever no ecrã e `scanf`() para ler dados do utilizador.
Sem essa biblioteca, o teu programa não conseguiria comunicar com o utilizador, incapaz de receber ou mostrar qualquer informação.

Para utilizar estas funções, é necessário incluir a biblioteca no início do programa, através de uma diretiva `#include`, de cabeçalho:

```
#include<stdio.h>
```

Esta diretiva informa o compilador de que serão usados os recursos definidos na biblioteca `stdio.h`. Sem essa inclusão, o programa não compila corretamente e não consegue interagir com o utilizador.
A seguir, apresenta-se algumas das principais funções da biblioteca `stdio.h`, que são utilizadas para operações de entrada e saída de dados.

Estas funções permitem a interação entre o programa e o utilizador, seja ao ler valores introduzidos pelo teclado, seja ao apresentar as informações no ecrã. Tais funções estão resumidas na Tabela 11 Funções de entrada e saída em C.

Tabela 11 Funções de entrada e saída em C

Função	Descrição breve	Uso principal
`printf()`	Escreve texto formatado no ecrã	Mostrar mensagens e valores
`scanf()`	Ler valores formatados do teclado	Ler números, caracteres e strings
`puts()`	Escreve uma string seguida de nova linha	Imprimir texto simples
`getchar()`	Ler um único carácter (requer Enter)	Ler teclas uma por uma
`fgets()`	Ler uma linha com segurança (limite de tamanho)	Ler frases com espaços
`sscanf()`	Ler dados de uma string formatada (como `scanf()`, mas com strings)	Extrair valores de uma string para variáveis
`gets()*`	Ler uma linha até Enter (sem controlo de tamanho)	Ler frases com espaços (evitar uso)
`getch()*`	Ler uma tecla sem mostrar no ecrã (sem esperar Enter)	Capturar teclas em menus ou jogos
`getche()*`	Igual ao getch(), mas mostra a tecla lida	Ler teclas visivelmente

***** Notas Importantes:

- A função `gets()` foi removida do padrão C11 por motivos de segurança. Recomenda-se o uso de `fgets()` em substituição.
- As funções `getch()` e `getche()` pertencem à biblioteca `conio.h`, que não é portátil e só funciona em alguns

compiladores. Evite usá-las em programas que devam ser compatíveis com diferentes sistemas.

Como já mencionado anteriormente, a inclusão é feita com a diretiva `#include`, seguida do nome da biblioteca, que é delimita-se por < >, por padrão, mas também se pode incluir delimitado por aspas duplas, entretanto este modo deve ser utilizado quando se inclui bibliotecas próprias, definidas pelo utilizador. O capítulo 14 trata deste assunto. Exemplo da inclusão de 2 bibliotecas, uma em cada linha:

```
#include<stdio.h>
#include <math.h>
// Sempre no início do código. É cabeçalho
```

Há muitas outras bibliotecas padrão na Linguagem C, algumas estão a ser apresentadas na Tabela 12.

Tabela 12 Algumas bibliotecas padrão em C

Biblioteca	**Função principal**
`stdio.h`	Entrada e saída de dados (`printf, scanf...`)
`stdlib.h`	Funções utilitárias (`malloc exit, atoi...`)
`math.h`	Funções matemáticas (`sqrt, pow, sin...`)
`string.h`	Manipulação de strings (`strlen, strcpy...`)
`ctype.h`	Testes e conversões de caracteres simples (`isalpha, toupper...`)

4.2 Especificadores de Formato

Os especificadores de formato são símbolos especiais usados em funções como `printf()` e `scanf()` para indicar o tipo de dado que está a ser impresso no ecrã ou lido do teclado. Tais especificadores possuem o carácter **%** seguidos por uma letra, que representa o tipo de informação.

Tais especificadores informam ao compilador como interpretar corretamente os dados que estão a ser manipulados. Usar um especificador inadequado pode causar erros de execução ou resultados incorretos. A Tabela 13 apresenta os especificadores de formato.

Tabela 13 Especificadores de formato em C (printf() e scanf())

Especificador	Tipo de dado	printf()	scanf()	Observações
`%c`	Carácter simples (`char`)	✅	✅	Em `scanf`, usar `" %c"` para evitar espaços/resíduos
`%d, %i`	Inteiro com sinal (`int`)	✅	✅	`%i` reconhece prefixos em `scanf()`
`%u`	Inteiro sem sinal (`unsigned`)	✅	✅	Para `unsigned int`
`%f`	Real (`float`)	✅	✅	Em `scanf`, `%f` lê `float`,
`%lf`	Real (`double`)	❌ ✅	✅	Em `scanf`, usar `%lf`. Em `printf`, tanto faz
`%e, %E`	Notação científica (`float`)	✅	❌	Exibe como 1.23e+03 ou 1.23E+03
`%g, %G`	Adaptativo (`float`)	✅	❌	Usa `%e` ou `%f`, o que for mais compacto
`%o`	Inteiro em octal	✅	✅	Em `scanf`, prefixo `0` já identifica octal

Especificador	Tipo de dado	printf()	scanf()	Observações
`%x, %X`	Inteiro em hexadecimal	✅	✅	Letras minúsculas (`a-f`) ou maiúsculas (`A-F`)
`%s`	String (texto)	✅	✅	`scanf` lê só até espaço. Para frases, usar `fgets()`
`%%`	Carácter `%`	✅	❌	Serve apenas para imprimir o % em `printf`

A seguir, apresenta-se um exemplo prático com código comentado que demonstra o funcionamento de alguns especificadores de formato. Este exemplo ilustra como declarar variáveis, ler diferentes tipos de dados do teclado e apresentar os valores formatados no ecrã.

```
// Lê e imprime um carácter
scanf(" %c", &letra);  // Exemplo: C
printf("%c\n", letra); // C
// Lê e imprime um inteiro
scanf("%d", &inteiro);   // Exemplo: -13
printf("%d\n", inteiro); //  -13
// Lê e imprime um inteiro sem sinal
scanf("%u", &positivo);   // Exemplo: 7
printf("%u\n", positivo); // 7
// Lê e imprime um número float
scanf("%f", &numFloat);   // Exemplo: 12.34
printf("%f\n", numFloat); // 12.340000
// Impressão em notação científica
printf("%e\n", numFloat); // 1.234000e+001
printf("%E\n", numFloat); // 1.234000E+001
// Lê e imprime um número double
scanf("%lf", &numDouble);   // Exemplo: 987.654321
printf("%lf\n", numDouble); // 987.654321
// Lê e imprime uma palavra
scanf("%s", palavra);      // Exemplo: Programar
printf("%s\n", palavra);  // Programar
// Impressão do símbolo %
printf("100%% completo!\n"); // 100% completo!
```

4.3 Comentários

Os comentários permitem inserir notas, explicações ou observações dentro do código-fonte, sem interferir na execução do programa. Comentar o código é uma boa prática de programação, pois melhora a legibilidade e facilita a manutenção por outros programadores (ou pelo próprio autor no futuro). Em C, existem dois tipos de comentários:

a. Comentário de uma linha: //: precede o comentário, mas limitado à própria linha

```
// Este é o comentário de uma linha
// se houver outra linha
// preceder por //
```

b. Comentário de várias linhas: /* */: tudo o que estiver entre os símbolos /* e */ é considerado comentário.

```
/* Este é o
   comentário em uma
   ou mais linhas   */
```

4.4 Caracteres especiais (escape) em printf()

Dentro de uma cadeia de caracteres (entre aspas duplas), se pode utilizar caracteres de escape, que começam com a barra invertida \. Estes códigos especiais permitem controlar a formatação do texto impresso no ecrã, inserindo quebras de linha, tabulações, aspas, entre outros. A Tabela 12 apresenta os principais caracteres de escape utilizados com `printf()`.

Tabela 14 Caracteres de escape

Código	Significado	Observação/Exemplo
`\n`	Nova linha	Quebra de linha (Enter)
`\t`	Tabulação horizontal	Espaço largo (equivale a Tab)
`\\`	Barra invertida	Mostra um \ literal
`\"`	Aspas duplas	Permite incluir aspas dentro da string
`\'`	Aspas simples	Para caracteres (`char letra = '\'';`)
`\b`	Backspace	Apaga o último carácter impresso
`%%`	Sinal de percentagem (%)	Único modo de mostrar % em `printf()`

A seguir se apresenta um exemplo de uso de alguns dos comandos apresentados na Tabela 14.

```
printf("Linha 1\nLinha 2\n"); // Quebra de linha
printf("Nome:\tMaria\n");     // Tabulação
printf("Caminho: C:\\pasta\\ficheiro\n");
// escrever barra invertida no ecrã
printf("Ele disse: \"Ola\"\n"); // escreve aspas
printf("Progresso: 100%%\n");   // escreve símbolo %
// Exemplo com vários caracteres especiais
printf("Nome:\tJoao\nIdade:\t25\nProgresso: 100%%\n");
```

A saída exibida no ecrã deste trecho de código é a seguinte:

```
Linha 1
Linha 2
Nome:   Maria
Caminho: C:\pasta\ficheiro
Ele disse: "Ola"
Progresso: 100%
Nome:   Joao
Idade:  25
Progresso: 100%
```

4.5 Saída de Dados com printf()

A função `printf()` é usada para apresentar mensagens e valores no ecrã durante a execução de um programa. Trata-se de uma das funções mais utilizadas da linguagem C, que mostra textos fixos e também o conteúdo de variáveis, com formatação personalizada.

O seu uso é essencial para comunicar com o utilizador, diagnosticar erros ou simplesmente acompanhar o comportamento do programa passo a passo.

Através dos especificadores de formato (como %d, %f, %c, %s etc.), `printf()` interpreta e exibe correctamente diferentes tipos de dados, como números inteiros, reais, caracteres e cadeias de texto.

Sintaxe:

```
printf("texto e formatos", listaVariáveis);
```

onde:

- `"texto e formatos"`: string que pode incluir texto fixo e especificadores de formato (como `%d, %f, %c, %s`).
- `listaVariáveis`: valores a serem impressos, na ordem dos formatos

Os especificadores mais usados são apresentados na Tabela 15.

Tabela 15 Exemplo de especificadores na função printf()

Formato	Tipo de dado	Exemplo
`%d`	Inteiro	`printf("%d", 42);`
`%f`	Real (ponto flutuante)	`printf("%f", 3.1415);`
`%c`	Carácter	`printf("%c", 'A');`
`%s`	String (texto)	`printf("%s", nome);`

4.6 Formatação de casas decimais

Ao usar `%f`, é possível fazer controlo do número de casas decimais com `.n`, onde **`n`** é o número de casas, por exemplo, se o número for 8.5:

`%.1f` → 1 casa decimal → 8.5

`%.2f` → 2 casas decimais → 8.50

`%.5f` → 5 casas decimais → 8.50000

A seguir se apresenta um exemplo simples e a respetiva saída

```c
int idade = 25;
printf("Idade: %d anos\n", idade);
// %d para inteiros

float nota = 8.5;
printf("Nota: %.1f\n", nota);
// %.1f para 1 casa decimal
```

```
Idade: 25 anos
Nota: 8.5
```

4.7 Entrada de Dados com scanf()

A função `scanf()` é utilizada para ler valores introduzidos pelo utilizador através do teclado.

Sintaxe:

```c
scanf("%tipo", &variavel);
```

É preciso ter atenção ao uso do símbolo `&` (e comercial) nas leituras com `scanf()`. Este símbolo é obrigatório antes da variável, pois indica ao programa o endereço de memória onde o valor lido será armazenado.

A única excepção são as strings (arrays de caracteres), que não utilizam o `&` ao serem lidas com `scanf()`.

No `scanf()`, também não se pode formatar o número de casas decimais como no `printf()`. Por exemplo, `scanf("%.2f", &altura);`, está errado e o programa não se comportará corretamente.

A função `scanf()` faz a leitura dos dados tal como são introduzidos. A formatação com casas decimais só é usada no `printf()`, no momento de mostrar os valores e não ao lê-los.

Um problema comum, ou seja, um erro frequente, ocorre quando se introduzem letras em vez de números ao usar `%d`. Nestes casos, o programa pode travar ou comportar-se de forma inesperada, uma vez que a função `scanf()` espera receber um valor numérico inteiro. Como o tipo de dado introduzido não corresponde ao esperado, a leitura falha.

Se este `scanf()` estiver dentro de uma estrutura de repetição, o programa pode entrar em *loop* infinito, pois o valor inválido permanece no buffer de entrada e continua a provocar falhas na leitura.

Mais adiante, serão vistas estratégias para validar a entrada e evitar este tipo de situação.

A seguir, apresenta-se um exemplo que mostra a leitura de diversos tipos de dados com a função `scanf()`. Este exemplo ilustra como ler inteiros, reais, caracteres e cadeias de caracteres de forma simples e correcta.

```c
printf("Nome: ");
scanf("%s", nome);  // Sem & porque é uma string
printf("Idade: ");
scanf("%d", &idade);  // Inteiro: com &
printf("Altura (em metros): ");
scanf("%f", &altura);  // Float: com &, sem formatação
printf("Sexo (M/F): ");
scanf(" %c", &sexo);
// Char: espaço antes de %c evita erro de leitura

printf("\n--- Dados Lidos ---\n");
printf("Nome: %s\n", nome);
printf("Idade: %d\n", idade);
printf("Altura: %.2f m\n", altura);
printf("Sexo: %c\n", sexo);
```

4.7.1 Leitura de Dados de uma String com sscanf()

Além de ler dados do teclado com `scanf()`, também é possível extrair informações diretamente de uma string já existente. Para isso, há a função `sscanf()`, que tem o mesmo formato de uso, mas recebe como primeiro argumento uma string de entrada.

A função `sscanf()` é especialmente útil quando se pretende extrair dados de uma cadeia de caracteres já existente. No capítulo 13 esta função será devidamente apresentada.

Sintaxe:

```
sscanf(string, "formato", lista_de_variáveis);
```

Entretanto, como exemplo de uso, veja o trecho de código a seguir

```c
char linha[] = "Bela 1.65 58";
char nome[20];
float altura;
int peso;

sscanf(linha, "%s %f %d", nome, &altura, &peso);

printf("Nome: %s\n", nome);      // Nome: Bela
printf("Altura: %.2f m\n", altura); // Altura: 1.65 m
printf("Peso: %d kg\n", peso);  // Peso: 58 kg
```

A Tabela 16 presenta um resumo das três formas mais comuns de leitura formatada em C (`scanf()`, `fscanf()` e `sscanf()`), organizada conforme a origem dos dados.

Tabela 16 Funções de leitura formatada conforme a origem dos dados

Função	Origem dos dados	Exemplo de uso
`scanf()`	Teclado	`scanf("%d", &x);`
`fscanf()`	Ficheiro	`fscanf(f, "%d", &x);`
`sscanf()`	String	`sscanf(str, "%d", &x);`

Com `sscanf()` é possível aplicar os mesmos especificadores de formato já apresentados. Além disso, se pode usar expressões mais avançadas como **`%[^,]`** (ler até encontrar uma vírgula) ou `%*` (ignorar partes da string). Exemplos mais complexos aparecem no capítulo sobre **strings**.

Veja a seguir exemplos do uso do `sscanf()` com o uso das formatações `%[^,]` e `%*`.

Exemplo 1: Ignorar a palavra "Capital" e extrair apenas o nome da cidade

```
char linha[] = "Portugal,Capital Lisboa";
char pais[20], cidade[20];

sscanf(linha, "%[^,],%*s %s", pais, cidade);

printf("Pais: %s\n", pais);      // Saida: Portugal
printf("Cidade: %s\n", cidade); // Saida: Lisboa
```

Breve explicação:

- `%[^,]` está a ler tudo até encontrar uma vírgula. Então armazena "Portugal" em `pais`.
- `%*s` ignora a primeira palavra depois da vírgula: `Capital`
- `%s` está a ler a palavra seguinte e armazena `Lisboa` em cidade.

Exemplo 2: Ignorar o domínio de um e-mail

```
char email[] = "joao.silva@gmail.com";
char utilizador[30];

// Lê até o caractere '@', ignorando o restante
sscanf(email, "%[^@]", utilizador);

printf("Nome de utilizador: %s\n", utilizador);
// Nome de utilizador: joao.silva
```

Breve explicação:

- `%[^@]` lê todos os caracteres até encontrar o símbolo `@`
- O restante da string (`gmail.com`) é ignorado automaticamente porque não há especificador para este trecho da string.

Exemplo 3: Ignorar a data e obter apenas o horário

```
char registo[] = "2025-08-05: 11:45";
char valor[6];

// Ignora a data e lê o horário
sscanf(registo, "%*[^:]: %s", valor);

printf("Horario: %s\n", valor);
//Horario: 11:45
```

Breve explicação:

– `%*[^:]` lê todos os caracteres até `:` e ignora (por causa do `*`)

– Depois de `:`, o `%s` lê o horário.

4.8 Exercícios Propostos

1. Cadastro Simples: Peça ao utilizador que introduza os seguintes dados:

- Nome (sem espaços)
- Idade (anos)
- Altura (metros)
- Peso (kg)

Depois, mostre os dados organizados no ecrã como:

Nome: _____

Idade: ___ anos

Altura: ___ m

Peso: ___ kg

2. Operações Aritméticas: Ler dois números inteiros e apresentar no ecrã os resultados das seguintes operações: Soma dos dois valores; Diferença entre o primeiro e o segundo valor lido; Produto dos dois valores; Divisão do primeiro valor pelo segundo (exibir o resultado com duas casas decimais)

3. Teste de Carácter e Número: Leia um carácter (com `%c`) e um número decimal (com `%f`) **.**

Faça o teste de introduzir letras, números ou frases como entrada e observe o que acontece.

Anote os resultados e identifique o que funciona ou não.

4. Leitura a partir de String: Dada a string: "`Maria 30 1.60 55.2`"

Use `sscanf()` para guardar: Nome (string), Idade (`int`), Altura (`float`) e Peso (`float`).

Mostre os valores lidos no ecrã.

5. Ignorar Palavras: Dada a string: "`Chave=9876`"

Use `sscanf()` para ignorar "`Chave=`" e guardar apenas o número 9876 numa variável inteira. Imprima o número.

6. Separar Cidade e Freguesia: Dada a string: "Lisboa, Freguesia Alvalade", use `sscanf()` com `%[^,]`, `%*s` e `%s` para:

- Guardar "Lisboa" na variável `cidade`
- Guardar "Alvalade" na variável `freguesia`

Mostre-os no ecrã.

7. Extrair o Domínio: Dada a string: "ana.rosa@gmail.com"

Use `sscanf()` para ignorar tudo **antes do caractere @** e guardar em utilizador. Mostre no ecrã:

```
Utilizador: gmail.com
```

8. Ler uma Frase Inteira: use `fgets()` para ler uma frase do utilizador (com espaços) e depois mostre essa frase no ecrã.

9. Desafio com sscanf()

Dada a string: "`Aluno: Carlos Andrade Silva`"

Use `sscanf()` e formatação para guardar apenas **`Silva`** numa variável `char` e mostrar no ecrã:

```
Último nome: Silva
```

5 *ESTRUTURAS CONDICIONAIS*

5.1 Introdução às Estruturas Condicionais

Na programação, muitas vezes é necessário tomar decisões com base em certas condições. Para isso, se utiliza as estruturas condicionais, que permitem ao programa executar ações diferentes consoante as circunstâncias. Um exemplo simples do dia a dia:

```
Se tiver dinheiro, então compro um café
Senão, bebo água
```

Este tipo de lógica também é usado nos programas:

```c
if (temDinheiro) {
    printf("Compre um café.\n");
} else {
    printf("Beba água.\n");
}
```

5.2 Instrução if()

A estrutura `if()` permite executar um bloco de código apenas quando uma determinada condição é verdadeira. Se a condição for falsa, o bloco é ignorado e o programa continua normalmente.

Sintaxe:

```
if (condição) {
    // Código executado se verdadeiro
}
```

Um exemplo simples, para verificar a idade de uma pessoa:

```
if (idade >= 18) {
    printf("Pode ter carta de condução!\n");
 }
```

As chavetas `{}` delimitam o bloco de comandos associado ao comando `if()` e são obrigatórias quando se quer executar mais de uma instrução.
Se houver apenas uma única instrução, as chavetas são opcionais, mas usar as chavetas é uma boa prática, pois evita erros e torna o código mais legível.

5.3 Instrução if() ... else

A instrução `if()...else` permite executar um bloco de código se uma condição for verdadeira (`if()`) e um bloco alternativo se a condição for falsa (`else`).
O comando `else` não tem condição própria, sendo apenas complementar ao `if()`. Se a condição do `if()` não for satisfeita, o código dentro do `else` é então executado.

Sintaxe:

```
if (condição) {
    // Bloco verdadeiro
} else {
    // Bloco falso
}
```

Um exemplo clássico é verificar se um número é par ou ímpar. Neste caso, se for digitado um número par, por exemplo, 14, executa o bloco associado ao `if()`, caso contrário, executa o bloco do `else`.

```
int numero;

printf("Digite um número inteiro: ");
scanf("%d", &numero);

if (numero % 2 == 0) {
    printf("O número é par.\n");
} else {
    printf("O número é ímpar.\n");
}
```

É muito importante compreender que o `else` **nunca** aparece sozinho, pois sempre está associado a um `if()` anterior. Além disso, o `else` não possui condição, penas define o que deve ser feito quando nenhuma das condições anteriores, associadas ao `if()`, for satisfeita.

5.4 Instruções Encadeadas (else if())

Em situações que exigem mais de duas possibilidades, pode-se usar estruturas de decisão encadeadas. Isso significa usar múltiplas instruções `if()` e `else if()` para testar diferentes condições, uma após a outra. O programa avalia cada condição na ordem em que aparecem e executa apenas o bloco correspondente à primeira condição verdadeira. Se nenhuma condição for satisfeita, o bloco `else` (opcional) é executado como alternativa final.

Uma estrutura `if()...else if()...else` é mais eficaz do que uma sequência de `if()` isolados, pois evita verificações desnecessárias assim que uma condição verdadeira é encontrada.

Este tipo de estrutura é útil para classificar valores, escolher opções em menus ou tratar diferentes faixas de entrada, dentre outra gama de aplicações.

```
if (nota >= 18) {
    printf("Excelente!\n");
} else if (nota >= 14) {
    printf("Bom!\n");
} else if (nota >= 10) {
    printf("Suficiente.\n");
} else {
    printf("Reprovado.\n");
}
```

As condições são testadas de cima para baixo. Assim que uma das condições for verdadeira, o bloco correspondente é executado e os demais são ignorados. O comando `else` no final do código trata o caso em que nenhuma das condições anteriores foi satisfeita.

5.5 Instrução switch() case

O comando `switch() ... case` é uma estrutura de controlo ideal para comparar igualdade de uma única variável contra vários valores possíveis.

Mas atenção, pois o `switch()...case` em C só funciona com variáveis de tipo inteiro ou carácter simples. É possível também comparar com `enum`, entretanto este conceito será apresentado no Capítulo 8. Além disso, ele apenas verifica igualdade e não permite usar qualquer operador relacionais.

A estrutura do `switch()…case` torna o código mais limpo e organizado, especialmente se há muitos possibilidades para uma mesma variável. Cada valor esperado é tratado num bloco `case` e pode-se usar, opcionalmente, a cláusula `default` para lidar com valores não previstos (tal como o `else` final em `if() else` aninhados).

Sintaxe:

```
switch (variável) {
    case valor1:
        // Código
        break;  // Sai do switch
    case valor2:
        // Código
        break;
    default:
        // Código se nenhum caso for válido
}
```

Neste exemplo a seguir, conforme o valor da variável `opcao`, o programa executa o bloco correspondente. O `default` é opcional e funciona como alternativa "caso nenhum dos valores anteriores seja igual".

```c
int opcao;
printf("1. Café\n2. Chá\n3. Sumo\nEscolha: ");
scanf("%d", &opcao);

switch (opcao) {
    case 1:
        printf("A preparar café...\n");
        break;
    case 2:
        printf("A ferver água para chá...\n");
        break;
    case 3:
        printf("A servir sumo...\n");
        break;
    default:
        printf("Opção inválida!\n");
}
```

5.6 Operador Ternário (? :)

O operador ternário `(?)` permite escrever expressões condicionais de forma mais compacta, substituindo pequenas estruturas `if() ... else`:

Sintaxe:

`condição ? valor_se_verdadeiro : valor_se_falso;`

Funciona como uma decisão em linha: se a condição for verdadeira, retorna o primeiro valor; caso contrário, retorna o segundo valor.

Neste exemplo a seguir, a variável maior recebe o valor de `a` se `a > b`, ou o valor de `b` caso contrário, ou seja, guarda o maior dos dois números.

```c
int maior = (a > b) ? a : b;
// Retorna o maior entre a e b
```

5.7 Exercícios Propostos

1. Elevador: Ler o peso total de pessoas num elevador. Se for superior a 450 kg, mostrar no ecrã a mensagem "Peso excedido!"

2. Velocidade e Multa: Ler a velocidade de um carro. Se for maior que 120 km/h, mostrar uma mensagem: "Multa por excesso de velocidade.". Caso contrário exibir: "Velocidade dentro do limite."

3. Calculadora de 2 Números (Simples): Ler dois números inteiros e uma operação (+, -, *, /), representada por uma opção de 1 a 4, onde 1 → Soma, 2 → Subtração, 3 → Multiplicação, 4 → Divisão.
Utilizar a estrutura switch para realizar a operação escolhida.
Se a opção for a divisão, apresentar o resultado com 2 casas decimais.
Atenção: Se o segundo número for zero e a operação for a divisão, mostrar uma mensagem de erro: `"Impossível dividir por zero."`

4. Parque de estacionamento grátis: Ler o valor de uma compra feita numa loja de um espaço comercial. Se o valor for maior ou igual a 50€, mostrar "Parque grátis", caso contrário, "Valor: 2€."

5. Desconto Progressivo: Ler o valor de uma compra e aplicar um desconto: até 100€ → sem desconto; de 101 a 200€ → 5%; acima de 200€ → 10%
Mostra a percentagem do desconto e o valor final a pagar.

6. Corrida de 100 metros: Ler o tempo (em segundos) de um corredor. Se for menor que 10 → "Atleta profissional!", se estiver entre 10 e 12 → "Muito bom!", entre 12 e 15 → "Regular" e maior de 15 → "Treines mais". Exibir a mensagem correspondente ao tempo. Lembre-se que não há tempo menor ou igual a zero.

7 – Dado da Sorte: Ler um número entre 1 e 6, que representa o resultado dum lançamento. Com base no valor, mostrar a mensagem correspondente:

- 6 → "Tiveste sorte máxima! Ganhas um prémio!"
- 5 → "Boa jogada! Avanças 3 casas."
- 4 → "Nada acontece. Esperas a próxima jogada."
- 3 → "Retrocedes 1 casa."
- 2 → "Perdes a vez."
- 1 → "Voltas ao início!"

Dica: Usar `switch()…case`.

8 – Combate de RPG: O teu personagem encontrou um inimigo! Ler o valor da força do teu herói (de 0 a 100) e a força do inimigo (de 0 a 100).
Comparar os valores e mostrar o resultado do combate:

- Se a força do herói for maior → "Vitória! O inimigo foi derrotado."
- Se for igual → "Empate! Ambos recuam."
- Se a força do inimigo for maior → "Derrota! Precisas treinar mais."

6 *ESTRUTURAS DE REPETIÇÃO*

6.1 Repetir bloco de comandos em C

Num programa, por vezes é necessário repetir o mesmo conjunto de instruções várias vezes. Em vez de copiar o mesmo código repetidamente, usam-se estruturas de repetição, que automatizam essa tarefa.

As estruturas de repetição permitem executar automaticamente um ou mais comandos, enquanto uma determinada condição se mantiver verdadeira.

Imagina que precisas de repetir uma mesma acção 100 vezes:

- Sem repetição automática: terias de escrever o mesmo código 100 vezes.
- Com repetição automática: bastam algumas linhas para fazer tudo de forma eficiente.

Esse tipo de repetição é normalmente chamado de ciclo, loop ou looping, onde todas estas denominações significam o mesmo: executar um bloco de código várias vezes, de forma controlada.

Estas estruturas tornam o programa mais compacto, organizado e fácil de manter.

6.2 Principais tipos em C

Agora que já sabes o que são ciclos, é o momento de conhecer os três principais tipos disponíveis na linguagem C. Cada um tem uma forma própria de funcionamento e é indicado para situações diferentes:

1. `while ()` → Executa um bloco de código enquanto uma condição for verdadeira. A verificação acontece antes da primeira execução, por isso, se a condição for falsa, à partida, o ciclo pode nem sequer iniciar.
2. `do...while()` → Neste caso, o bloco é executado pelo menos uma vez, pois a condição é verificada após a primeira execução.
3. `for()` → Usado principalmente quando já se sabe à partida quantas vezes o código deverá ser repetido. É ideal para contagens controladas, como percorrer de 1 a 10.

Nos tópicos a seguir se apresenta como funciona cada uma destas estruturas. Em todos os casos, saiba que a condição deve tornar-se falsa em algum momento, caso contrário o ciclo nunca termina, ou seja, pode causar um *loop infinito*.

6.3 Estrutura while()

1. Verifica a condição.
2. Se verdadeira executa o bloco e volta ao passo 1.

Este ciclo se repete até que a condição se torne falsa

Sintaxe:

```
while (condição) {
    // Código repetido
}
```

Exemplo: Contagem regressiva de 5 a 1 seguida da mensagem “Fogo”. Este exemplo mostra como usar a estrutura `while()` para criar uma contagem decrescente.

```c
int i = 5;
while (i >= 1) {
    printf("%d...\n", i);
    i--;  // ou i -= 1; ou i = i - 1;
}
printf("Fogo!\n");
```

```
5...
4...
3...
2...
1...
Fogo!
```

Cuidado! Se a condição de paragem nunca for satisfeita, o ciclo torna-se um *loop* infinito e o programa continuará a executar sem parar, o que pode levar ao consumo excessivo de recursos do sistema ou obrigar o utilizador a encerrar manualmente a execução.

6.4 Estrutura do...while ();

1. Executa o bloco de código uma vez.
2. De seguida, verifica a condição.
3. Se a condição for verdadeira, repete o ciclo a partir do passo 1.

Este ciclo garante pelo menos uma execução, mesmo que a condição seja falsa logo no início.

Sintaxe:

```
do {
    // Código
} while (condição);
```

Pedir ao utilizador que introduza uma palavra-passe e encerrar o programa assim que a mesma estiver correta. O programa repete o pedido enquanto a palavra-passe inserida for inválida, garantindo que apenas termina quando o valor introduzido corresponder ao esperado.

```
int senha;
do {
    printf("Digite a palavra passe de 4 numeros: ");
    scanf("%d", &senha);
} while (senha != 1234);
printf("Acesso permitido!\n");
```

6.5 Estrutura for()

1. Inicia uma variável de controlo
2. Verifica a condição: se for verdadeira executa o bloco
3. Após o bloco, atualiza a variável (incremento ou outro)
4. Volta ao passo 2

Este ciclo é ideal quando se conhece antecipadamente o número de repetições.

Sintaxe:

```
for (inicio; condicao; passo) {
    // Código
}
```

Onde:

- `inicio`: executa apenas uma vez, antes de começar o ciclo.
- `condicao`: verifica antes de cada repetição; se falsa, encerra o ciclo
- `passo`: executado ao final de cada repetição (exemplo: incremento).

Exemplo 1: Imprimir números pares de 0 a 10, conforme a saída apresentada:

```
for (int i = 0; i <= 10; i += 2) {
    printf("%d ", i);
}
```

```
0 2 4 6 8 10
```

Exemplo 2: Imprimir os inteiros de 5 até 0 conforme a saída apresentada:

```
for (int i = 5; i >= 0; i--) {
    printf("%d\n", i);
}
```

```
5
4
3
2
1
0
```

Para fins de compreensão e comparação, eis aqui um trecho de código com `while` (), onde se apresenta um código equivalente a este anterior:

```
int i = 5;
while (i >= 0) {
    printf("%d\n", i);
    i--;
}
```

Uma consideração importante é que todos os argumentos da função `for()` são opcionais, ou seja, cada um dos três argumentos (início, condição, passo) pode ser omitido, a depender da lógica do programa. Entretanto, o símbolo `;` que separa os argumentos deve ser obrigatoriamente mantido. Veja a seguir:

1. Sem o argumento "início"

A variável pode ser iniciada antes do `for()` .

```c
int i = 0;
for (; i < 3; i++) {
    printf("%d\n", i);
}
```

2. Sem a "condição"

Forma um *loop* infinito, precisa de `break`, associado a um `if()` para parar.

```c
for (int i = 0;; i++) {
    if (i == 3) {
        break;
    }
    printf("%d\n", i);
}
```

O comando `break` interrompe imediatamente a execução de um ciclo de repetição (`while`, `do...while` ou `for`), mesmo que a condição do ciclo ainda seja verdadeira.

No entanto, o `break` só faz sentido dentro de uma estrutura de seleção (`if`, por exemplo), já que deve existir uma condição que determine *quando* o ciclo deve ser interrompido.

Este comando é útil quando se quer sair de um ciclo com base numa situação específica que ocorre durante a execução.

3. Sem o "passo" no for()

O incremento pode ser feito dentro do bloco.

```c
for (int i = 0; i < 3;) {
    printf("%d\n", i);
    i++;  // passo manual
}
```

4. Sem "início" e "passo"

Desta forma o comando `for()` fica muito similar ao comando `while()`.

```
int i = 0;
for (; i < 3;) {
    printf("%d\n", i);
    i++;
}
```

5. Com todos os argumentos omitidos

Loop infinito (como usar `while(1)`) exige `break` para parar.

```
for (;;) {
    // ciclo infinito
   if (condicao){
     break;
   }
}
```

Um exemplo simples, para exibir os valores de 0 até 4 está a ser apresentado a seguir.

Uma modificação possível é, se quiseres exibir de 1 até 4, basta trocar o comando de pós incremento `(i++)`, dentro do `printf()`, para o comando de pré-incremento `(++i)`.

```
int i=0;
for (;;) {
   if (i==4){
     break;
   }
   printf("i = %d\n",i++);
   //primeiro exibe, depois incrementa de 1
}
```

```
i = 0
i = 1
i = 2
i = 3
```

Observações:

- Mesmo com os campos dos argumentos vazios, os pontos e vírgulas devem ser mantidos para garantir a sintaxe correta
- Usar `for()` com omissões é comum em programas mais avançados, mas deve-se evitar exageros para manter o código legível.

6.6 Comandos de Controlo

6.6.1 break

O comando `break`, anteriormente apresentado neste mesmo capítulo, é usado para interromper o *loop* imediatamente. Normalmente está associado a uma condição dentro do laço de repetição.

Exemplo: O loop encerra ao digitar 0

```
while (1) {  // Loop infinito
 int num;
 printf("Digite um número (0 para sair): ");
 scanf("%d", &num);
 if (num == 0) {
     break;  // Sai do loop
 }
 printf("Digitaste: %d\n", num);
 }
```

6.6.2 continue

O comando `continue` é utilizado dentro de ciclos de repetição para ignorar o restante do bloco de código e passar imediatamente para a próxima iteração do ciclo.

Sempre que o `continue` é executado, o programa salta as instruções seguintes dentro do bloco e volta ao início do ciclo para avaliar novamente a condição.

Este comando deve ser usado dentro de uma estrutura de seleção, como um `if()`, que determina em que situação o salto será feito.

Exemplo de um código em que o número 3 não é exibido no ecrã, porque quando `i == 3`, o `continue` fez com que o programa saltasse diretamente para a próxima iteração do `for`, sem executar o `printf()`.

```
for (int i = 1; i <= 5; i++) {
 if (i == 3) {
     continue;  // Salta a iteração se i é igual a 3
 }
 printf("%d ", i);
 }
```

6.7 Repetições Aninhadas

Repetições aninhadas (ou ciclos dentro de ciclos) são especialmente úteis em situações onde é necessário percorrer estruturas bidimensionais, como matrizes, tabelas ou combinações de valores.

O conceito é simples: um ciclo é colocado dentro do corpo de outro ciclo, formando um bloco repetitivo mais complexo. O ciclo interno será executado completamente a cada vez que o ciclo externo avança uma iteração.

Este tipo de construção funciona com qualquer tipo de repetição (`for`, `while` ou `do...while`) desde que devidamente estruturada.

Repetições aninhadas são amplamente usadas em aplicações como manipulação de arrays bidimensionais (matrizes); geração de tabelas ou

padrões repetitivos; simulação de combinações ou pares ordenados de valores; criação de menus com subopções, entre outros.
É importante manter a lógica clara e garantir que cada ciclo tenha sua própria variável de controlo, evitando conflitos ou comportamentos inesperados.

Um exemplo simples e direto é a exibição da tabuada de 1 a 5, apresentando os resultados de cada multiplicação de forma organizada.
Este tipo de exercício é útil para praticar ciclos aninhados (*loops* dentro de *loops*) e reforçar o entendimento sobre estruturas de repetição em C.

```c
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 10; j++) {
        printf("%d x %d = %d\n", i, j, i * j);
    }
    printf("\n");
}
```

A saída terá este padrão para todos os valores de 1 até 5

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10
```

se repete até 5

```
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Veja como é a execução, passo a passo, do código

1. O ciclo externo (`for (int i = 1; i <= 5; i++)`) controla qual número da tabuada será exibido.

- Inicia com `i = 1` e vai até `i = 5`.

2. Para cada valor de `i`, o ciclo interno (`for (int j = 1; j <= 10; j++)`) executa 10 vezes, calculando a multiplicação `i * j`.
   - O `j` vai de `1 a 10`, formando as linhas da tabuada do número atual `i`.
3. A função `printf()` exibe a multiplicação no formato: `i x j = resultado`
4. Depois de mostrar a tabuada completa de um número (`j` foi até `10`), o comando `printf("\n")` adiciona uma linha em branco para separar da próxima tabuada.

6.8 Exercícios Propostos

1. Números de 1 a 100: utilize o comando **`for()`** e depois o comando **`while()`** para imprimir no ecrã todos os números inteiros de 1 até 100.

2. Média de 10 Números: Ler 10 valores inteiros do teclado e calcular a média destes números lidos.
Dica: apesar dos valores serem inteiros, a média é um número real (`float`). Somente após somar todas os valores é que podes calcular a média, fora da estrutura de repetição.

3. Números Primos: Verificar se um número inteiro lido do teclado é primo, ou seja, se este número é divisível apenas por 1 e pelo próprio número. Exibir o uma mensagem: `É primo` ou `Não é primo`.

4. Fatorial: Calcular o fatorial de um número (por exemplo, se o número inserido pelo utilizador for 5: 5! = 5 × 4 × 3 × 2 × 1 = 120). Exibir o fatorial calculado, ao final.

5. Adivinhar um número: O computador tem um número secreto entre 1 e 100. O utilizador deve tentar adivinhar. A cada tentativa, o programa diz se o palpite é maior ou menor que o número secreto. O jogo termina quando o número for adivinhado.

Dica: a função `rand()` da biblioteca `<stdlib.h>` gera números aleatórios. Para evitar que os mesmos valores se repitam sempre, inicializar com `srand()` usando o tempo atual, da biblioteca `<time.h>`:

```
#include <stdlib.h>
#include <time.h>
...
    //exemplo
    srand(time(NULL)); // Inicia com base no tempo do sistema
    int numero = rand() % 100 + 1; // Gera número entre 1 e 100
```

- `rand()` gera um número aleatório
- `% 100` limita de 0 a 99
- `+1` ajusta para 1 a 100

Sem `srand()`, a sequência será sempre a mesma.

6. Jogo: Escape do Labirinto: O jogador está preso num labirinto e precisa encontrar a saída. A cada jogada, o utilizador escolhe uma direção: N (norte), S (sul), L (leste) ou O (oeste). A saída está numa direção específica secreta, definida aleatoriamente pelo programa. O jogo continua até que o utilizador acerte a direção certa. Regras e dicas:

- Usar um `do...while()` para repetir as tentativas.
- Usar `rand()` para escolher a direção secreta (`N, S, L` ou `O`).
- Dar mensagens de tentativa ("Parede! Tenta outra.")
- Ao fim, mostrar quantas tentativas foram feitas
- Atenção: letras maiúsculas são diferentes de minúsculas em C. Ao comparar caracteres simples, lembra-te de colocá-los entre **aspas simples**, como 'N', 's' ou 'L'.

Exemplo de interação:

```
Estas preso num labirinto misterioso...
Tentes encontrar a saida!

Digites uma direcao (N/S/L/O): o
Parede! Tentes outra direcao.
Digites uma direcao (N/S/L/O): l
Encontraste a saida em 2 tentativa(s)!
```

7. Jogo de Desafio de Dados - Criar um programa onde dois jogadores virtuais lançam dados para disputar quem obtém o maior valor.

Regras:

- O utilizador escolhe se quer usar dados de 6 faces (1) ou dados de 12 faces (2).
- Em cada jogada, o Jogador 1 e o Jogador 2 lançam um dado.
- O programa mostra os valores sorteados para cada jogador e anuncia o `vencedor da ronda.`
- Se os dois tiverem o mesmo valor, é considerado empate.
- O jogo repete-se enquanto o utilizador quiser.

Dicas:

- Usar `rand()` para gerar os valores aleatórios.
- Utilizar `do...while()` para repetir o jogo.
- Comparar os valores com `if/else` para determinar o vencedor.
- Permitir ao utilizador jogar quantas vezes quiser, perguntando ao final de cada jogada se deseja continuar.

Exemplo de interação:

```
Bem-vindo ao jogo de dados!

Quantos dados queres jogar? (1 ou 2): 1
Jogador: 4
Computador: 5
Computador venceu esta ronda.

Desejas jogar novamente? (s/n): s

Quantos dados queres jogar? (1 ou 2): 2
Jogador: 3 e 1 (Total: 4)
Computador: 2 e 1 (Total: 3)
Venceste esta ronda!

Desejas jogar novamente? (s/n): n
Obrigado por jogar!
```

7 ARRAYS

7.1 Arrays Unidimensionais (Vetores)

Arrays (ou vetores) são coleções de variáveis do mesmo tipo, armazenadas de forma sequencial na memória. Os arrays guardam diversos valores relacionados sob um único nome, de forma a facilitar a organização e o acesso aos dados.

Cada elemento do array é identificado por um número chamado índice, que indica a sua posição dentro do vetor.

Na linguagem C, os índices começam sempre em zero (0). Isso significa que o primeiro elemento está na posição 0, o segundo na posição 1 e assim sucessivamente. Portanto, um vetor com **n** elementos terá índices que vão de 0 até n – 1.

Conforme representado Figura 7-1

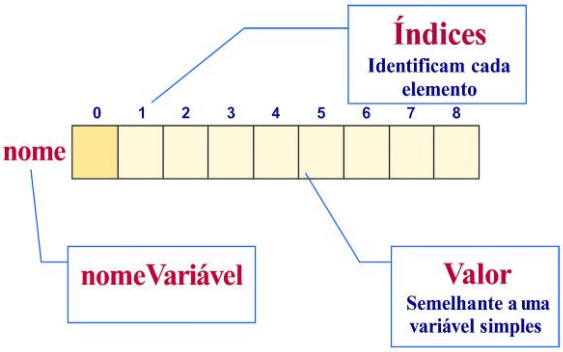


Figura 7-1 - Array unidimensional (ou Vetor)

Não existe vetor sem índice, pois para aceder a qualquer elemento é obrigatório a respetiva posição do valor do elemento.

Para melhor compreensão, considere um prédio com vários andares, onde cada apartamento está numerado a partir do 0 (rés do chão). O nome do prédio (ou o próprio número da porta) é o nome do vetor e o número do andar corresponde à posição (índice) de cada elemento.

Exemplo

```
int predio[3];          // Prédio com 3 pisos
predio[0] = 00;         // Apartamento no rés do chão
predio [1] = 10;         // Apartamento do 1.º andar
predio [2] = 20;         // Apartamento do 2.º andar
```

Neste exemplo anterior, a primeira linha de código permite perceber como se faz a declaração de um array, cuja sintaxe está a seguir.

```
<tipo> nome_do_vetor[tamanho];
```

Exemplos de declaração:

```
int idade[5];       // Idade de 5 pessoas
float altura[10];   // Altura de 10 pessoas
char letra[26];     // String de tamanho 26
```

Para aceder aos elementos se usa os respetivos índices, que começam invariavelmente em 0. A contagem nunca se inicia em 1.

```
// Primeiro elemento
idade[0] = 10;
// Último elemento: índice 4 num vetor de tamanho 5
idade[4] = 50;
```

7.1.1 Leitura e Escrita em arrays

Após declarar um array, torna-se necessário preencher os seus elementos com valores e, posteriormente, aceder a esses mesmos valores. Em C, tanto a leitura como a escrita em arrays são realizadas através dos índices, geralmente com o auxílio de ciclos `for()`, mas pode ser através de qualquer estrutura de repetição.

O exemplo seguinte declara um array com cinco posições e utiliza um ciclo `for()` para solicitar ao utilizador que introduza cinco idades. Em seguida, outro ciclo é utilizado para apresentar os valores armazenados.

Se fizeres a leitura (`scanf()`) e a impressão (`printf()`) dos dados dentro do mesmo ciclo `for()`, a saída no ecrã se tornará confusa, dificultando a compreensão do que está a acontecer.

Para melhor perceberes este comportamento, recomenda-se testar o código num compilador C: primeiro tal como se apresenta a seguir, com dois ciclos `for()`, e depois numa versão alterada com apenas um ciclo `for()` para se fazer a leitura e escrita.

```
int idade [5];
// Preencher o vetor
for (int i = 0; i < 5; i++) {
    printf("Digite a %d° idade: ", i+1);
    scanf("%d", &idade[i]);
}
// Exibir o vetor
for (int i = 0; i < 5; i++) {
    printf("%d ", idade[i]);
```

Tenhas atenção, pois um erro muito comum ao trabalhar com vetores é tentar aceder a uma posição fora dos seus limites.

Por exemplo, ao declarar um vetor com 5 elementos, `int idade[5];`, os índices válidos vão de 0 a 4 ou seja, com 5 posições no total.
No entanto, é comum o principiante pensar que pode usar `idade[5]`, o que está fora do intervalo permitido. E aceder a `idade[5]` não gera erro visível imediato, mas o programa pode apresentar comportamento imprevisível, como ler valores errados, escrever em zonas de memória indevidas ou até falhar completamente. Este é considerado um erro lógico grave.
Sempre que declarares um vetor com `n` posições, lembre-se que os índices vão de `0` até `n - 1`.

7.1.2 Inicialização de Vetores

Em C, um vetor pode ser preenchido de várias formas: após a declaração, através de atribuições manuais ou de ciclos for, ou ainda logo no momento da declaração, com valores definidos entre chaves.
Esta última forma, conhecida como inicialização directa, é compacta, clara e segura, pois garante que os elementos do vetor começam com valores bem definidos, evitando erros comuns causados por variáveis não inicializadas.
Considere estes exemplos a seguir.

```
// Tamanho explícito. Início manual dos elementos
int pares[5] = {2, 4, 6, 8, 10};

/* Tamanho automático: o compilador conta os
   elementos (3 neste caso) */
int impares[] = {1, 3, 5};

// Todos os 50 elementos iniciados a zero
int numeros[50] = {0};
```

No primeiro exemplo, o vetor `pares` tem tamanho fixo e os seus elementos são atribuídos directamente, por ordem. No segundo, o vetor `impares` não indica o tamanho; o compilador deduz automaticamente com base no número de valores fornecidos.

O terceiro exemplo mostra uma técnica útil para garantir que todas as posições de um vetor começam com um mesmo valor, no caso o valor `0`. Este método é útil quando o vetor será usado como contador ou acumulador. Para além disso, também se pode preencher um vetor com zero ou qualquer outro valor, utilizando um ciclo `for()`:

```
for (int i = 0; i < 100; i++) {
    dados[i] = 0;
 }
```

Ambas as abordagens são válidas e podem ser escolhidas conforme a situação e a clareza desejada no código.

7.2 Matrizes (Arrays Bidimensionais)

Um array pode ter mais do que uma dimensão. Os arrays unidimensionais, ou vetores, foram apresentados no Tópico 7.1, onde cada elemento é acedido através de um único índice.

Apesar de ser possível declarar arrays com qualquer número de dimensões (n-dimensionais), este livro limita-se ao estudo de arrays com no máximo duas dimensões: os arrays bidimensionais, também designados por matrizes. As matrizes utilizam dois índices para aceder aos seus elementos, um índice para indicar a linha e outro índice para a coluna. Os elementos encontram-se

organizados num formato tabular, distribuídos nas tais linhas e colunas, conforme ilustrado na Figura 7-2.

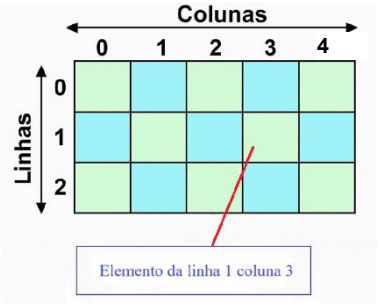


Figura 7-2 Matrizes ou arrays com 2 dimensões

Da mesma forma que se fez com os vetores, pode se usar a analogia com um prédio, mas neste caso, com vários apartamentos por andar. Imagines um prédio com diversos andares, onde cada andar tem muitos apartamentos.
O nome ou o número da porta do prédio representa o nome da matriz, tal como aconteceu com o vetor. No entanto, para identificar um apartamento, é necessário indicar duas informações: o andar e o número do apartamento nesse andar.

Veja este exemplo a seguir.

```
// Prédio com 3 pisos, 4 apartamentos por andar
int predio[3][4];
```

Neste caso, declara-se uma matriz com 3 linhas (pisos) e 4 colunas (apartamentos por piso). Os índices válidos para esta matriz são:

- Linhas: de 0 a 2 (3 piso)
- Colunas: de 0 a 3 (4 apartamentos por piso)

Cada elemento da matriz é identificado por dois índices e acede-se aos elementos (apartamentos) da seguinte forma:

```
predio[linha][coluna];
```

Então, para este exemplo:

```
predio[0][0] = 01; // Apartamento 1 no rés do chão
predio[0][1] = 02; // Apartamento 2 no rés do chão
predio[1][2] = 12; // Apartamento 2 no 1º andar
predio[2][3] = 23; // Apartamento 3 no 2º andar
```

A sintaxe para declarar matrizes é a seguinte:

```
<tipo> nome_da_matriz[linhas][colunas];
```

Exemplos:

```
int tabela[3][5];      // 3 linhas, 5 colunas
float notas[4][4];     // Matriz quadrada, 4X4
char mapa[10][20];     // 10 linhas, 20 colunas
```

7.2.1 Leitura de matrizes

Tal como acontece com os vetores, também nas matrizes é possível percorrer os elementos utilizando ciclos. No entanto, como a matriz tem duas dimensões, é necessário usar dois ciclos aninhados: um para as linhas e outro para as colunas.

Para ler os valores de uma matriz a partir do teclado, percorrem-se todas as posições com dois ciclos `for()`.

Neste exemplo a seguir, o programa percorre as 3 linhas e 4 colunas da matriz e pede ao utilizador que introduza os valores de cada elemento, um a um.

```
int matriz[3][4];
int i, j;

for (i = 0; i < 3; i++) {
    for (j = 0; j < 4; j++) {
        printf("Valor da posição [%d][%d]: ", i, j);
        scanf("%d", &matriz[i][j]);
    }
}
```

7.2.2 Escrita de uma matriz

Para mostrar os valores da matriz no ecrã, o processo é semelhante: usam-se dois ciclos `for()` para percorrer cada elemento e apresentar o seu conteúdo.

```
for (i = 0; i < 3; i++) {
    for (j = 0; j < 4; j++) {
        printf("%d ", matriz[i][j]);
    }
    printf("\n");
}
```

Este código mostra a matriz com o mesmo formato de uma tabela, com uma linha por cada ciclo externo.

Este método aplica-se a qualquer matriz com duas dimensões. Basta ajustar os limites dos ciclos conforme o número de linhas e colunas da matriz.

```c
int numeros[2][3];
// Preencher a matriz
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        printf("Digite o valor [%d][%d]: ", i, j);
        scanf("%d", &numeros[i][j]);
      }
  }

 // Exibir a matriz
 for (int i = 0; i < 2; i++) {
   for (int j = 0; j < 3; j++) {
       printf("%d\t", numeros[i][j]);
       // %d\t - numero inteiro separado por um
       // distanciamento igual a tabulação
   }
   printf("\n");  //pular linha após imprimir todos
                  //elementos daquela linha em questão
 }
```

É sempre importante frisar que, tal como nos vetores, os índices começam em 0. Se declarar uma matriz com 3 linhas e 4 colunas (`int m[3][4]`), os índices válidos variam de `[0][0]` até `[2][3]`.

Tentar aceder um elemento com `m[3][0]` ou `m[0][4]`, fora dos limites da matriz, pode ocasionar um comportamento imprevisível, como leitura de lixo da memória, sobrescrita de dados ou falhas no programa.

7.2.3 Como é a execução

Pela sua versatilidade e clareza estrutural, as matrizes constituem uma das ferramentas fundamentais no desenvolvimento de soluções organizadas e eficientes. É, por isso, essencial compreender o seu funcionamento no contexto da programação.

As matrizes são amplamente utilizadas em aplicações que exigem representação espacial, como jogos (como, sudoku ou xadrez), interfaces gráficas, mapas, processamento de imagens e também em cálculos científicos e matemáticos, tais como sistemas de equações, transformações lineares e algoritmos de álgebra matricial.

Já se apresentou anteriormente que para trabalhar com todos os elementos de uma matriz, usa-se dois ciclos `for()` aninhados: um para percorrer as linhas e outro para percorrer as colunas.

A forma mais comum é percorrer linha por linha, ou seja:

- o primeiro `for()` percorre as linhas
- o segundo `for()` percorre as colunas dentro de cada linha

Considere o seguinte trecho de código e a respetiva explicação de seu funcionamento.

```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 3; j++) {
        printf("matriz[%d][%d]: ", i, j);
        scanf("%d", &matriz[i][j]);
    }
}
```

Este código preenche a matriz da seguinte forma:

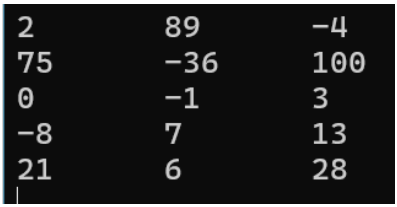
- Quando `i = 0`, o programa entra no `for` das colunas (`j`) e percorre `j = 0`, `j = 1` e `j = 2`. Isso significa que preenche toda a linha 0 (primeira linha da matriz)

- Quando `j = 3`, a condição `j < 3` deixa de ser verdadeira e o ciclo das colunas termina — então o programa volta ao ciclo das linhas e `i` é incrementado para 1, iniciando a próxima linha
- Portanto, `i = 1` e preenche a linha 1, ao entrar novamente no `for` das colunas (`j`) e percorre `j = 0`, `j = 1` e `j = 2`
- E assim sucessivamente para todas as demais linhas.

Para exibir a matriz no ecrã, considere o código a seguir e a respetiva explicação de funcionamento.

```
// Exibir (formato de tabela):
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d\t", matriz[i][j]);
        // \t para alinhar
    }
    printf("\n");
}
```

Este segundo trecho percorre a matriz novamente com dois ciclos aninhados. Para cada linha `i`, o programa percorre as colunas `j`, imprimindo os valores com `\t` (tabulação horizontal), de modo a alinhá-los em colunas. No final de cada linha, imprime-se uma quebra de linha `(\n)` para que a matriz apareça organizada como uma tabela no ecrã.
Um exemplo de como esta matriz está a ser exibida no ecrã está a ser apresentado a seguir:



7.3 Exercícios Propostos

1. Soma dos elementos de um vetor: Ler um vetor com 10 números inteiros e calcular a soma de todos os seus elementos. Exibir esta soma.

2. Contar números pares: Ler um vetor com valores inteiros e contar quantos desses valores são números pares. Exibir os valores e o total.

3. Elementos da diagonal principal: Ler uma matriz 3 × 3 de números inteiros e exibir no ecrã apenas os elementos da diagonal principal (onde o número da linha é igual ao da coluna).

4. Somar linhas e colunas: Ler uma matriz 3 × 3 de reais e calcular a soma dos elementos de cada linha. Calcular também a soma dos elementos de cada coluna. Exibir o somatório das linhas e também das colunas.

5. Somar linhas e colunas e guardar em vetores separados: Ler uma matriz de valores inteiros com dimensão 3 × 3. A seguir, gerar e exibir no ecrã estes dois vetores:

– No vetor denominado `linha`, em cada posição, guarda a soma dos elementos da respetiva linha da matriz

– No vetor `coluna`, em cada posição, guarda a soma dos elementos da respetiva coluna.

6. Subtrair com base na diagonal principal: Ler uma matriz, por exemplo A, de valores reais, com dimensão 3 × 3. Gerar uma nova matriz B, onde cada elemento de B é igual à subtração entre o elemento correspondente da matriz A e o elemento da diagonal principal que se encontra na respetiva linha. Ao final, exibir ambas as matrizes.

7. Gerar matriz com padrão fixo: Gerar e exibir uma matriz 6 × 6 que segue o seguinte padrão:

1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3

8. Contar peças num tabuleiro de xadrez: Ler uma matriz 8 × 8 de caracteres, que representa um tabuleiro de xadrez. Cada casa (elemento) pode conter apenas estes caracteres:

'P' para peão	'C' para cavalo	'R' para rei
'T' para torre	'B' para bispo	'D' para dama
		'-' para casa vazia

O programa deve contar e apresentar quantas peças de cada tipo existem no tabuleiro no momento atual do jogo.

Dica: usar `scanf(" %c", &matriz[i][j]);` (note o espaço antes do `%c`) para ler corretamente os caracteres.

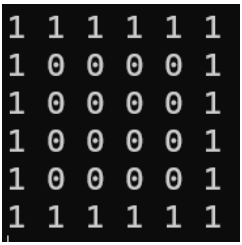
9. Mapa com obstáculos: Criar uma matriz 10 × 10 preenchida com `'.'` (ponto), que está a representar um mapa vazio. Depois, ler do utilizador 5 diferentes posições (linha e coluna) e colocar um 'X' em cada uma, de forma a representar obstáculos. Por fim, imprimir o mapa resultante.

10. Localização de tesouro: Gerar uma matriz 5 × 5 com '−' em todas as posições. Depois, ler a posição de um tesouro (linha e coluna) e marcar com 'T'. Ler também a posição de um jogador e marcar com 'J'. Imprimir o tabuleiro. Não permitir que o jogador e o tesouro estejam na mesma posição.

11. Cruz no centro: Criar uma matriz 7 × 7 onde todas as posições com índices da linha central e da coluna central recebem o valor 1, e todas as outras recebem 0. Imprimir a matriz.

```
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
1 1 1 1 1 1 1
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
```

12. Bordas com valor diferente: Criar uma matriz 6 × 6 onde todas as posições da borda (primeira e última linha, primeira e última coluna) recebam o valor 1 e o restante da matriz o valor 0. Por fim, exibir a matriz.



13. Detetar linha cheia (como em jogos do tipo Tetris.)**:** Ler uma matriz 5 × 5 de inteiros, que contém apenas 0 ou 1. Verificar se existe alguma linha totalmente preenchida com 1. Caso exista, exibir o número dessa linha.

8 CADEIAS DE CARACTERES: STRINGS

Em muitas aplicações, é necessário trabalhar com texto, sejam nomes, mensagens, palavras ou frases. Em C, esse tipo de informação é armazenado através de cadeias de caracteres, também conhecidas como strings.

Ao contrário de outras linguagens modernas, a linguagem C não possui um tipo string nativo. Em vez disso, uma string é representada por um vetor unidimensional do tipo `char`, terminado obrigatoriamente pelo carácter especial *\0*, que indica o final da cadeia.

Este carácter nulo (`\0`) não é visível quando se imprime a string, mas é essencial para que as funções da linguagem saibam onde termina o texto.

8.1 Declaração duma string

A sintaxe para declarar uma string é semelhante à de um vetor de caracteres:

```
char nome[tamanho];
```

Por exemplo, nesta declaração `char nome[20];`, a variável `nome` pode armazenar até 19 caracteres visíveis, mais o carácter `\0` ao final. Ou seja, está declaração separa espaço de memória para guardar uma palavra ou frase curta com, no máximo, 19 letras.

8.2 Leitura de Strings

Em C, há diversas formas de ler texto do utilizador. Cada uma tem comportamentos específicos, vantagens e desvantagens.

A Tabela 1Tabela 17 resume os métodos mais comuns:

Método	Comportamento	Limitações / Observações
`scanf("%s", nome)`	Ler até o primeiro espaço em branco.	Não lê frases; trunca no primeiro espaço.
`fgets(nome, tamanho, stdin)`	Ler até Enter (`\n`) ou até atingir o tamanho.	Inclui o `\n` na string (pode ser removido).

Tabela 17 Comparação entre métodos de entrada de strings

Uma observação importante é que a função `gets()` foi usada durante muito tempo para ler strings completas (incluindo espaços), mas é considerada insegura. Isso porque não verifica o tamanho da string, o que pode causar sobrescrita de memória (*buffer overflow*) e falhas no programa.
Por esse motivo, `gets()` foi removida do padrão da linguagem C (C11) e não deve ser utilizada. A alternativa segura é `fgets()`, que permite limitar a quantidade de caracteres lidos e evita erros graves.

8.2.1 Leitura com fgets()

Um exemplo de leitura com `fgtes()` está apresentado a seguir.

```
char nome[30];
printf("Digite o seu nome completo: ");
fgets(nome, 30, stdin);
/* Ler da entrada padrão (stdin - standard input)
   até 29 caracteres + '\0' */
```

A função irá ler até um carácter de nova linha (\n) ou até atingir o tamanho máximo de 29 caracteres). Importante saber que após essa leitura, a variável

`nome` conterá o texto digitado incluindo o `\n` final, desde que haja espaço suficiente para armazená-lo.

Se por exemplo o utilizador digitar: `Ana Maria`

Como há espaço suficiente no array, a variável nome conterá: `Ana Maria\n\0`

Este detalhe é importante porque, ao trabalhar com a string (comparar, imprimir, calcular tamanho, etc.), o \n pode influenciar o resultado.

A. Como remover o \n

Uma técnica comum para remover o carácter de nova linha (se estiver presente) é:

```
int i = 0;
while (nome[i] != '\0') {
    if (nome[i] == '\n') {
        nome[i] = '\0';
        break;
    }
    i++;
}
```

Esta rotina percorre a string e substitui o primeiro `\n` encontrado por `\0`, eliminando o salto de linha.

B. Se não há espaço suficiente na string

Pode ocorrer a situação em que o utilizador introduz uma cadeia de caracteres com mais elementos do que o tamanho definido. Considere, por exemplo, que para a string definida anteriormente com tamanho 30, o utilizador digite o seguinte nome: `Ana Maria Rodrigues Santos Oliveira.`

A função `fgets()` vai ler no máximo 29 caracteres, pois o último é reservado para o `\0`. O restante da string ficará na entrada padrão (`stdin`), à espera da próxima leitura, pois como não houve espaço para armazenar o \n, este comando também ficará pendente, o que pode causar comportamento anormais em leituras seguintes.

Depois da leitura com `fgets(nome, 30, stdin);`, a variável `nome` conterá: `Ana Maria Rodrigues Santos Ol\0`.

O resto da frase — `iveira\n` — fica guardado na entrada e pode ser lido na próxima chamada a `fgets()`.

Dentre as consequências, pode-se citar:

- A string fica truncada: o utilizador acredita que digitou tudo, mas a variável só tem parte do texto
- A linha seguinte do programa pode não ter o conteúdo verdadeiro, pois pega o ler o resto que ficou no `stdin`
- O utilizador tem a sensação que o programa "saltou" uma leitura, especialmente se estiveres a usar `scanf`() logo depois.

Ao usar `fgets()`, é fundamental escolher um tamanho adequado para a string e estar atento à presença do `\n`. Se o texto digitado for maior do que o tamanho permitido, a leitura será incompleta e o restante ficará na entrada, podendo interferir com as próximas leituras.

8.2.2 Leitura com scanf()

A função `scanf()` pode ser usada para ler strings com o especificador de formato `%s`, como no exemplo a seguir:

```
scanf("%s", nome);
```

Este comando está a ler apenas até ao primeiro espaço em branco, ou seja, só captura a primeira palavra. O espaço, o *tab* ou o *Enter* são considerados separadores de entrada. Portanto, se o utilizador digitar: `Ana Maria`, a variável `nome` conterá apenas: `Ana\0`. O restante da string, `Maria\n`, permanece na entrada padrão (`stdin`) e poderá ser capturado por comandos de entrada seguintes.

A. Limitações do scanf("%s", ...)

- Não permite a leitura de frases com espaços
- Pode causar comportamento inesperado se for usado após `fgets()`, ou vice-versa
- Pode causar comportamento inesperado se for usado após `scanf()`
- Não há forma directa de limitar a leitura ao tamanho do vetor, embora seja possível prevenir problemas usando um modificador de tamanho:

```
scanf("%29s", nome); // ler até 29 caracteres + '\0'
```

Contudo, mesmo com essa limitação, o `scanf("%s", ...)` permanece com o problema de não ler o espaço e não guardar na string o conteúdo após o primeiro separador. Por isso, não é recomendado quando se pretende ler nomes completos, frases ou qualquer cadeia que contenha espaços.

B. Recomendações

- Para ler apenas uma palavra, o `scanf("%s", ...)` pode ser suficiente
- Para ler frases ou nomes completos, deve-se preferir o uso de `fgets();`
- Nunca alternar ou ler em ciclos, `scanf()` e `fgets()` sem limpar a entrada, independentemente do tipo de dados lido. O `\n` deixado por `scanf()` pode ser imediatamente capturado por `fgets()` e causar leituras vazias.

8.3 Funções da Biblioteca <string.h>

Uma string em C é, na verdade, um vetor de `char` terminado por `'\0'`. Por isso, também é possível percorrer, ler ou modificar uma string usando estruturas como o `for()`, acedendo cada carácter individualmente com índices.

No entanto, as funções da biblioteca `<string.h>` tornam essas operações muito mais simples, rápidas e seguras evitando erros comuns e tornando o código mais limpo.

A biblioteca `<string.h>` fornece várias funções para manipulação de strings em C. Essas funções facilitam tarefas como copiar, comparar, obter o tamanho, juntar ou modificar o conteúdo de cadeias de caracteres. Na Tabela 18 estão algumas das funções mais utilizadas:

Tabela 18 Algumas funções da biblioteca string.h

Função	Descrição
`strlen(s)`	Retorna o comprimento da string s (sem o `\0`)
`strcpy(dest, src)`	Copia o conteúdo de `src` para `dest`
`strncpy(dest, src, n)`	Copia até `n` caracteres
`strcat(dest, src)`	Junta (concatena) `src` no final de `dest`
`strcmp(s1, s2)`[1]	Compara duas strings (0 se forem iguais)
`strncmp(s1, s2, n)`	Compara os primeiros **n** caracteres
`strchr(s, c)`	Retorna apontador para o primeiro `c` encontrado em `s`
`strstr(s1, s2)`	Retorna apontador para a primeira ocorrência de `s2` em `s1`
`strcspn(s1, s2)`	Conta quantos caracteres em `s1` antes de encontrar qualquer um de `s2`
`strtok(s, delimita)`	Divide a string `s` em partes (tokens), separadas por caracteres `delimita` (o delimitador)
`strupr(s)`[2]	Converte todos os caracteres da string para MAIÚSCULAS
`strlwr(s)` [2]	Converte todos os caracteres da string para minúsculas

Observação:

[1] A função `strcmp()` compara duas strings carácter por carácter em ordem lexicográfica (alfabética). A função não compara o tamanho (quantidade de caracteres), mas sim o conteúdo e a ordem dos caracteres na tabela ASCII.

- Retorna 0 → se as strings forem iguais
- Retorna < 0 → se a primeira string for menor
- Retorna > 0 → se a primeira string for maior

Exemplo:

```
strcmp("Ana", "Bruno");     // retorna < 0
strcmp("Maria", "Maria");   // retorna 0
strcmp("Zé", "Ana");        // retorna > 0
strcmp("Maria", "maria");   // retorna < 0 porque
            //maiúsculas vêm antes de minúsculas
```

[2] As funções `strupr()` e `strlwr()` não fazem parte do padrão ANSI C, mas estão disponíveis em muitos compiladores.

Para programas que precisam ser portáveis entre diferentes plataformas, recomenda-se não usar essas funções diretamente. Em vez disso, pode-se implementar versões próprias utilizando a biblioteca `<ctype.h>`, que oferece funções para converter carácter a carácter (como `toupper()` e `tolower()`).

As principais funções da biblioteca `<ctype.h>` serão tratadas no item 8.4. seguir, apresenta-se um breve exemplo de utilização da função `strcmp()`.

```
char senha[20];
printf("Digite a palavra passe: ");
fgets(senha, 20, stdin);
// Comparar com admin e incluir \n
if (strcmp(senha, "admin\n") == 0) {
    printf("Acesso permitido!\n");
} else {
    printf("Senha incorreta.\n");
}
```

Agora que já conheces algumas funções da biblioteca `<string.h>` e que também já sabes que, por vezes, é necessário remover o carácter `\n` no final da string, podes fazê-lo com a função `strcspn()` da biblioteca `<string.h>`, como no exemplo a seguir.

```
fgets(cidade, sizeof(cidade), stdin);
cidade[strcspn(cidade, "\n")] = '\0';
```

Neste código, o `fgets()` está a ler a string e armazena o `\n` ao final, se houver espaço suficiente. O comando `strcspn(cidade, "\n")` devolve a posição do primeiro `\n` encontrado na string e essa posição é então usada para substituir o `\n` por `'\0'`, encerrando corretamente a string.
Esse procedimento evita que a string contenha uma quebra de linha invisível, garantindo que operações posteriores funcionem como esperado.

8.4 Extração Formatada com sscanf()

Em muitas situações práticas, é necessário extrair dados estruturados de dentro de uma string, como quando ao ler uma linha de texto de um ficheiro ou de entradas formatadas de uma API, ficheiro .txt ou CSV. A função `sscanf()` resolve exatamente este tipo de problema.
Esta função é como uma versão de `scanf()` que opera sobre uma string em memória, em vez de ler do teclado.
Sintaxe:

```
sscanf(str, formato, variáveis);
```

onde:

- `str`: string a ser interpretada
- `formato`: especificadores como em `scanf`
- `variáveis` onde os valores serão armazenados

Neste primeiro exemplo a seguir, esta função `sscanf()` está a ser usada para extrair dados separados por delimitadores

```
char linha[] = "X=120, Y=30";
int x, y;
sscanf(linha, "X=%d, Y=%d", &x, &y);
printf("X: %d\nY: %d\n", x,y);
```

```
X: 120
Y: 30
```

Já neste segundo, os dados são separados por `|`:

```
char dados[] = "Nome: Filipa Maria| Idade: 27";
char nome[50];
int idade;

sscanf(dados, "Nome: %49[^|]| Idade: %d", nome, &idade);
// %49[^|] está a ler até o carácter |
printf("%s tem %d anos\n",nome, idade);
```

```
Filipa Maria tem 27 anos
```

As Tabela 19 resume as funções mais utilizadas para ler ou processar dados a partir de strings.

Tabela 19 Funções de leitura de strings

Função	Origem dos dados	Quando usar
`scanf()`	Teclado	Entrada interativa do utilizador
`fscanf()`	Ficheiro (ficheiro)	Ler de ficheiros com dados estruturados
`sscanf()`	String (memória)	Processar texto já lido ou recebido

8.5 Tipo enumerado (enum)

Em C, o tipo enumerado (`enum`) permite definir um conjunto de valores inteiros nomeados. É útil quando se quer representar um grupo de opções fixas com nomes significativos, como dias da semana, tipos de utilizador ou estados de um processo. A sintaxe é simples:

`enum nomeVariavel { valores};`

por exemplo:

```
enum Dia { SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA };
```

Cada nome definido numa enumeração representa um valor inteiro sequencial. Por padrão, a contagem começa em 0, tal como os índices de qualquer array: `SEGUNDA vale 0; TERCA vale 1, ..., SEXTA vale 4.`

Mas, por não ser array, se o programador achar necessário, é possível escolher manualmente o valor inicial. Por exemplo:

```
enum Dia { SEGUNDA = 1, TERCA, QUARTA, QUINTA, SEXTA };
```

Neste caso, a enumeração começa em 1 e os valores seguintes seguem automaticamente: `SEGUNDA vale 1; TERCA vale 2, ..., SEXTA vale 5.`

8.6 Funções da Biblioteca <ctype.h>

A biblioteca `<ctype.h>` fornece funções para testar e converter caracteres individuais. Essas funções são muito úteis para verificar se um carácter é

letra, número espaço, maiúscula etc. ou mesmo converter caracteres de forma segura, um por um, dentro de strings.

A Tabela 20 apresenta algumas funções da biblioteca `ctype.h`.

Todas essas funções recebem um valor `char` (ou um `int` representando um carácter) e retornam um valor inteiro (geralmente 0 ou diferente de zero).

Tabela 20 Algumas funções da biblioteca ctype.h

Função	**O que faz**	**Exemplo**
`isalpha(c)`	Verifica se c é uma letra (A–Z, a–z)	`isalpha('A') → verdadeiro`
`isdigit(c)`	Verifica se c é um dígito (0–9)	`isdigit('5') → verdadeiro`
`isalnum(c)`	Verifica se c é letra ou dígito	`isalnum('X') → verdadeiro`
`isspace(c)`	Verifica se c é espaço, tab ou quebra	`isspace(' ') → verdadeiro`
`isupper(c)`	Verifica se c é letra maiúscula	`isupper('F') → verdadeiro`
`islower(c)`	Verifica se c é letra minúscula	`islower('f') → verdadeiro`
`toupper(c)`	Converte c para maiúscula, se for letra	`toupper('a') → 'A'`
`tolower(c)`	Converte c para minúscula, se for letra	`tolower('A') → 'a'`

Tabela 16 – Algumas Funções da biblioteca <ctype.h>

Essas funções são especialmente úteis ao validar entradas do utilizador, fazer filtros ou converter strings manualmente carácter a carácter. Podem ser usadas em conjunto com `for()`, para percorrer uma string completa.

Ao contrário das funções `strupr() e strlwr()` (não padronizadas), `toupper() e tolower()` são padronizadas e fazem parte da linguagem

C desde o início, sendo seguras de usar para converter carácter a carácter para maiúsculo ou minúsculo, respetivamente.

O exemplo a seguir apresenta um exemplo de uso de um destes comandos.

```
char palavra[] = "programacao";
//Percorrer a string e converter em maiúscula
for (int i = 0; palavra[i] != '\0'; i++) {
    palavra[i] = toupper(palavra[i]);
}
printf("Em maiúsculas: %s\n", palavra);
```

A linha do for pode ser substituída por:

```
for (int i = 0; i< strlen (palavra); i++) {
```

Ou por qualquer outro comando de repetição.

Há muitas aplicações possíveis, como por exemplo, analisar uma string e contar quantas letras maiúsculas ela contém. Esta tarefa é comum em validações de palavras-passe ou em análises de texto.

O exemplo a seguir, está a percorrer cada carácter de uma string e verificar se se trata de uma letra maiúscula. Para isso, utiliza-se a função `isupper()` da biblioteca `<ctype.h>`, que devolve verdadeiro sempre que o carácter analisado for uma letra entre `'A' e 'Z'`.

```c
char texto[] = "Programar em C";
int maiusculas = 0;
for (int i = 0; texto[i] != '\0'; i++) {
    if (isupper(texto[i])) {
        maiusculas++;
    }
}
printf("Total de letras maiúsculas: %d\n", maiusculas);
```

8.7 Limpeza do buffer de entrada

Ao alternar a leitura de números e caracteres em C, é comum que o carácter de nova linha (`\n`), gerado ao pressionar Enter, permaneça no buffer de entrada (`stdin`). Esse carácter não é consumido por funções como `scanf()` e pode ser lido acidentalmente por uma leitura seguinte do tipo `char` ou `char[]`, causando erros inesperados.

Uma solução frequentemente usada é `fflush(stdin)`, mas o seu uso com a entrada padrão não é definido pela norma C e o comportamento pode variar entre sistemas (funciona no Windows, mas pode falhar no Linux, por exemplo).

A forma mais segura e recomendada é utilizar `getchar()` para remover o `\n` residual antes da próxima leitura de `char`, garantindo que o buffer fique limpo.

```c
scanf("%d", &idade);
getchar(); // remove o '\n' pendente
```

O uso do `getchar()` é a solução mais aceita devido a maneira simples e direta de consumir (e descartar) caracteres residuais do buffer, bem como ter

controlo sobre o que está sendo removido, geralmente o `\n`. Para além disso, há a portabilidade, pois funciona em qualquer sistema e compilador, garantindo consistência no comportamento do programa.

Existem ainda outras técnicas para limpar o buffer manualmente, utilizando recursos do próprio `scanf()`. Por exemplo:

```
scanf("%*[^\n]");   // descarta tudo até o '\n'
scanf("%*c");       // descarta o próprio '\n'
```

Essas duas instruções combinadas eliminam todo o conteúdo restante da linha atual, sendo úteis quando se quer garantir que o buffer esteja limpo antes de uma nova leitura de string ou carácter.

Compreender o funcionamento do buffer de entrada e aplicar essas práticas é essencial para evitar falhas silenciosas em programas interativos e garantir uma experiência de entrada correta e confiável.

8.8 Exercícios Propostos

1. Contador de Vogais: Ler uma string de até 100 caracteres e contar quantas vogais (a, e, i, o, u, maiúsculas e minúsculas) existem. Ao final exibir a quantidade total de vogais encontradas.

2. Inversor com Espaços Mantidos: Ler uma string de até 80 caracteres e exibir a mesma string invertida, mas mantendo os espaços nas posições originais.

3. Filtro de Dígitos: Ler uma string de até 80 caracteres e criar outra string apenas com os dígitos encontrados na string original. Exibir ambas as strings.

4. Capicua (ou palíndromo) Simples: Ler uma palavra de até 50 caracteres e verificar se ela é uma capicua (lida igual da esquerda para a direita e vice-versa). Exibir uma mensagem correspondente.

5. Contagem de Palavras: Ler uma frase de até 100 caracteres e contar quantas palavras existem (separadas por espaços). Ao final exibir o número total de palavras.

6. Ocultar Informações Numéricas: Ler uma frase de até 120 caracteres e substituir todos os dígitos por *. Exibir ao final a frase resultante com números ocultados.

7. Comparação Sem Diferença de Maiúsculas: Ler duas palavras, cada uma com até 30 caracteres e verificar se são iguais. Ignorar maiúsculas/minúsculas. Ao final exibir a mensagem: "Iguais" e a string ou "Diferentes" e ambas as strings.

8. Mini Forca: Armazene uma palavra secreta com no máximo 20 caracteres. O utilizador deve tentar adivinhar a palavra, em até 10 tentativas, introduzindo uma letra de cada vez. Como saída no ecrã, após cada tentativa, mostrar o estado da palavra, usando o carácter _ para as letras ainda não descobertas. Ao final, apresentar uma mensagem: jogador venceu ou perdeu.

```
=== Mini Jogo da Forca ===
Adivinhe a palavra! Tens 10 tentativas.

Palavra: _____
Tentativas restantes: 10
Digite uma letra: a
Letra 'A' nao encontrada!

Palavra: _____
Tentativas restantes: 9
Digite uma letra: e

Palavra: _E__E
Tentativas restantes: 9
Digite uma letra: t

Palavra: TE_TE
Tentativas restantes: 9
Digite uma letra: s

Parabens! Acertastes: TESTE
```

9. Cifra de César (Deslocamento +3): Ler uma frase de até 80 caracteres e deslocar cada letra 3 posições à frente no alfabeto (A → D, B → E, ..., X → A, Y → B, Z → C (mesmo para minúsculas). Os demais caracteres (espaços, pontuação, números) devem permanecer inalterados.

Ao final exibir a frase codificada.

```
Digite uma frase: A zebra no zoo
Frase codificada: D cheud qr crr
```

10. Tabuleiro com Palavra Oculta: Crie uma matriz 10 × 10 preenchida com letras maiúsculas aleatórias. Escolha ou leia uma palavra de até 10 caracteres e insira-a horizontalmente numa linha aleatória do tabuleiro, com início numa coluna aleatória que permita que a palavra caiba no espaço disponível.

Como saída se deve exibir no ecrã o tabuleiro completo e a indicar a posição inicial da palavra (linha e coluna) onde ela foi inserida.

```
Digite uma palavra (max. 10 letras): Programar

Tabuleiro:
G U W N S G T F M S
I P R O G R A M A R
H I G D B E A C T M
R C B V N S P R S Y
E K B F K V O J M L
R C E X Z A Y I X O
N O C H B L W W H R
S X B D Y V R T B E
Y X E R C P X R G Z
S G J V X F U G C Q

Palavra inserida na linha 1, coluna 1.
```

Dicas:

- Use a função `rand()` para gerar números aleatórios entre 0 e 25, convertendo-os em letras com 'A' + valor
- Lembre-se de iniciar o gerador aleatório com `srand(time(NULL));` no início do `main()`
- Para escolher a posição inicial, use `rand()` limitado pelo tamanho da matriz e da palavra
- Certifique-se de que a palavra não ultrapasse o limite da linha escolhida
- Pode usar `toupper()` da biblioteca `<ctype.h>` para converter a palavra para maiúsculas antes de inseri-la no tabuleiro

11. Extração de Campos: Ler uma string no formato "nome;idade;profissão", com até 80 caracteres, e separar os dados. Exibir cada dado em uma linha, com o respetivo rótulo:

Nome: …

Idade: …

Profissão: …

9 FUNÇÕES

9.1 A Função main()

Antes de apresentar como são as funções em C, é fundamental compreender o papel especial da função principal, o `main()`.

Todo programa em C começa a sua execução pela função `main()` e é a partir desta que o computador segue as instruções definidas pelo programador. Quando esta função termina, todo o programa também termina.

Além disso, um programa só pode ser compilado com sucesso e gerar um ficheiro executável se contiver a função `main()`, pois esta é o ponto de entrada obrigatório.

O `main()` pode, tecnicamente, ter qualquer tipo de retorno ou até ser declarada como `void`, como em:

```
void main() {
    // corpo da função
}
```

No entanto, a forma mais comum e recomendada é usar `int` como tipo de retorno, indicando, ao sistema operativo, a finalização do programa. Vale ressaltar que embora `void main()` seja aceito por alguns compiladores, não é padrão ANSI C. O uso de `int main()` é o mais portátil e adequado em programas profissionais.

```c
int main() {
    // Código do programa
    return 0;  // Indica término com sucesso
}
```

O valor 0 no `return` representa uma execução bem-sucedida. Outros valores podem indicar diferentes tipos de erro, se necessário.

A função `main()` pode chamar outras funções, desde que estas tenham sido declaradas (protótipos) ou definidas antes da chamada. A ordem em que as funções aparecem no código não determina a ordem de execução, pois o que importa é a ordem em que são chamadas.

Por isso, um programa com dezenas de funções pode ser executado de forma organizada, chamando cada uma na sequência definida dentro do `main()` ou de outras funções.

9.1.1 Argumentos de linha de comando

A função main() também pode receber argumentos da linha de comandos, permitindo que o programa receba dados logo no momento em que é executado.

Para isso, utiliza-se a seguinte forma de declaração:

```c
int main(int argc, char *argv[]) {
    // ...
}
```

Onde

- `argc` (*argument count*) indica o número de argumentos passados ao programa.
- `argv` (*argument vector*) é um vetor de *strings* contendo esses argumentos, sendo `argv[0]` o nome do próprio programa e os elementos seguintes (`argv[1]`, `argv[2]`, …) os argumentos fornecidos pelo utilizador.

Este recurso é útil, por exemplo, para criar programas que processam ficheiros indicados pelo utilizador ou que recebem parâmetros de configuração diretamente na execução.
Um exemplo deste uso pode ser pensado em um programa que recebe um nome de ficheiro como argumento e o abre diretamente. Na execução, pode-se escrever no terminal:

```
meuprograma dados.txt
```

Neste caso:

- `argc` tem valor `2`
- `argv[0]` contém "`meuprograma`"
- `argv[1]` contém "`dados.txt`"

Com este recurso, é possível criar programas mais flexíveis, capazes de se adaptar a diferentes entradas sem recompilação.

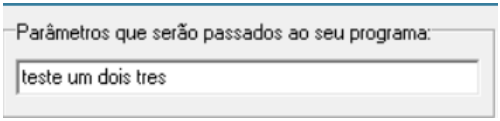
A seguir, há um exemplo curto, simples e objetivo, com comentários para facilitar a compreensão:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    // Mostra quantos argumentos foram passados
    printf("Total de argumentos: %d\n", argc);

    // Mostra cada argumento recebido
    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    return 0;
}
```

Após guardar como `teste.c`, compilar, executar assim no terminal:

```
./teste um dois três
```

Ou no sítio específico do teu compilador, como por exemplo:



A saída esperada é a seguinte, com a diferença apenas no diretório que guardaste o código.:

```
Total de argumentos: 5
argv[0] = C:\_Ines\teste.exe

argv[1] = teste

argv[2] = um

argv[3] = dois

argv[4] = tres
```

9.2 Por Que Usar Funções

Funções permitem dividir um programa em partes menores, com tarefas bem definidas. Isso torna o código mais organizado, legível e fácil de manter.
Ao agrupar comandos num único bloco reutilizável evita-se repetições desnecessárias e facilita a resolução de erros ou melhorias futuras. Cada função pode ser escrita uma vez e usada sempre que for necessário.
Dentre as vantagens, cita-se algumas consideradas importantes:

- Organização: Divide o programa em blocos lógicos, facilitando a leitura e compreensão
- Reutilização: Evita escrever o mesmo código várias vezes
- Manutenção: Permite fazer alterações numa parte específica do código sem afetar o restante
- Testabilidade: Cada função pode ser testada separadamente.

9.3 Passagem de Parâmetros: Valor vs Referência

Para começar é importante entender que as funções podem receber informações do programa principal através de parâmetros. Esses parâmetros podem ser passados de duas formas principais:

- **Por valor:** é enviada uma cópia do dado. A função pode modificar o valor copiado, mas isso não altera a variável original fora da função.
- **Por referência:** é enviada a referência (endereço) da variável. A função pode alterar o valor diretamente na memória. Essa técnica exige o uso de apontadores.

Neste capítulo se apresenta a passagem de parâmetro por valor, que é mais simples e segura para iniciantes.

A passagem de parâmetro por referência será explicada com mais detalhes no Capítulo 11, após a introdução a apontadores no Capítulo 10.

9.4 Estrutura Básica de uma Função

Uma função em C é composta por quatro partes principais:

a) **Tipo de retorno**: Indica o tipo de valor que a função vai devolver (por exemplo, `int`, `float`, `char`, `void`)

b) **Nome da função:** Escolhido pelo programador, seguindo as regras para nomes em C

c) **Parâmetros:** Entre parênteses, após o nome da função. São os dados que a função pode receber. Se a função não receber nenhum dado, utiliza-se `void`

d) **Bloco de código** – Entre chavetas, { }, contém as instruções que a função executa.

```
tipo <nomeFuncao> (lista_de_parametros)    {
    // Instruções
    return valor;
 }
 // OBS.: Se o tipo de retorno for void, basta return;
// ou se pode omitir a linha do return
```

9.5 Protótipo de função

Antes do `main()`, é comum declarar um protótipo da função, indicando apenas a sua assinatura:

```
tipo <nomeFuncao> (lista_de_parametros);
```

O protótipo termina com ponto e vírgula (;) e essa é a principal diferença em relação à definição completa da função, ou seja, do cabeçalho da função.
Os nomes das variáveis nos parâmetros podem ser omitidos no protótipo. Apenas os tipos são obrigatórios.
O protótipo informa ao compilador que existe uma função com aquele nome e formato. Isso permite chamar a função na `main()` mesmo que a sua definição só apareça depois da chamada da função.

9.6 Tipos de Funções e Chamada

Depois de declarada e definida, uma função pode ser chamada a partir do `main()` ou de outras funções.
A chamada de uma função consiste em escrever o nome da função seguido de parênteses, com ou sem argumentos, conforme o caso.
Em C, pode-se criar diferentes tipos de funções combinando:

- **Com ou sem retorno:** se a função está ou não a devolver um valor
- **Com ou sem parâmetros:** se a função está a receber ou não dados externos à função.

A Tabela 21 mostra como uma mesma tarefa, que é somar dois números, pode ser implementada de formas diferentes, a depender do tipo da função, da forma como os dados são passados ou o resultado é devolvido.

Vale ressaltar que cada uma dessas versões das funções são correctas e funcionais, mas oferece graus diferentes de flexibilidade. Por exemplo, as funções com parâmetros podem ser usadas para somar quaisquer valores enquanto as versões sem parâmetros estão limitadas aos números definidos internamente.

Tabela 21 Tipos de funções e respetivos exemplos simples

Tipo da Função	Exemplo Simples

Sem retorno
Sem parâmetros
	`void somar(void) {`
	`  int a = 2, b = 3;`
	`  printf("%d\n", a + b)`
	`}`

Sem retorno
Com parâmetros
	`void somar(int a, int b) {`
	`  printf("%d\n", a + b);`
	`}`

Com retorno
Sem parâmetros
	`int somar(void) {`
	`  int a = 2, b = 3;`
	`  return a + b;`
	`}`

Com retorno
Com parâmetros
	`int somar(int a, int b) {`
	`  return a + b;`
	`}`

A Tabela 22, a seguir, mostra como as funções definidas na Tabela 21 podem ser chamadas no programa principal (`main()`), dependendo do seu tipo e da sua estrutura.

Tabela 22 Exemplo de chamadas de função

Tipo da Função	Chamada no main()
Sem retorno Sem parâmetros	`somar();`
Sem retorno Com parâmetros	`somar(2, 3);`
Com retorno Sem parâmetros	`int r = somar();` `printf("%d\n", r);`
Com retorno Com parâmetros	`int r = somar(2, 3);` `printf("%d\n", r);`

Observações:

- Quando a função tem retorno, o resultado pode ser guardado numa variável ou usado diretamente no `printf()`.
- Quando não tem retorno, a função apenas realiza uma ação, como imprimir o valor no ecrã.
- As funções com parâmetros são mais versáteis, pois permitem utilizar diferentes valores em cada chamada.

Uma prática comum em programas simples é escrever todas as funções antes da função `main`

Dessa forma, como o compilador lê o código de cima para baixo, ele já conhece a existência e a definição dessas funções no momento em que elas são chamadas, dispensando o uso de protótipos.

A seguir se apresenta um exemplo desta prática.

```c
void saudacao() {
    printf("Olá!\n");
}

int main() {
    saudacao();
    return 0;
}
```

Nesse caso, como a função `saudacao()` foi definida antes da `main()`, o compilador já sabe que esta função existe aquando for chamada.

No entanto, essa prática, embora funcione, não é considerada ideal em programas maiores ou organizados em vários ficheiros. A boa prática é separar a declaração (protótipo) da função da sua implementação completa, permitindo uma estrutura de código mais clara, modular e reutilizável.
O uso de protótipos de função, geralmente colocados antes da `main()` ou num ficheiro de cabeçalho (`.h`), permite chamar funções mesmo que sua definição completa só apareça depois no código.

9.7 Escopo de Variáveis

O escopo de uma variável define onde esta variável pode ser usada e por quanto tempo permanece acessível durante a execução do programa.
Em C, o escopo é determinado pelo local onde a variável é declarada e existem dois tipos principais: local e global.

9.7.1 Escopo Local

Uma variável de escopo local só pode ser usada dentro do bloco onde foi declarada, sendo normalmente, dentro de uma função.

Assim, quando o programa sai dessa função, a variável deixa de existir e o espaço de memória que ocupava é liberado.

No exemplo a seguir, a variável `x` só existe dentro da função `exemplo()`. Se houver alguma tentativa de usá-la fora da função, o compilador gerará um erro, pois `x` não é reconhecida fora do seu bloco.

```
void exemplo() {
      int x = 10;  // x é local a esta função
      printf("%d\n", x);
}
```

Além do escopo local nas funções, também é possível declarar variáveis dentro de blocos menores, como estruturas de decisão (`if()`), ciclos (`for()`, `while()`) ou até mesmo chavetas isoladas `{ ... }`.

```
if (1) {
    int a = 20;  // visível só dentro do if
    printf("%d\n", a);
}
// printf("%d", a);  // ERRO: fora do escopo
```

9.7.2 Escopo Global

Uma variável de escopo global é declarada fora de qualquer função, normalmente no início do programa.

```c
int global = 5;  // escopo global

void mostrar() {
    printf("%d\n", global);
}
```

Ela pode ser usada por qualquer função no mesmo ficheiro e, se declarada com a palavra-chave `extern`, pode até ser partilhada entre diferentes ficheiros do projeto.

Mas tenha atenção, pois embora variáveis globais sejam úteis em alguns casos, é boa prática preferir variáveis locais sempre que possível. Isso ajuda a evitar conflitos, torna o código mais modular e reduz o risco de erros difíceis de rastrear.

9.7.3 Reutilização de Nomes de variáveis

Também é possível usar o mesmo nome de variável em escopos diferentes, como por exemplo uma variável global e uma variável local com o mesmo nome. Nesses casos, a variável local se sobrepõe à variável global dentro da função onde foi declarada.

Isso acontece porque a memória das variáveis é organizada como uma pilha (*stack*) e quando a função é chamada, as variáveis locais são empilhadas sobre as variáveis já existentes. Ao final da função (ou do bloco) essas variáveis locais são automaticamente removidas (desempilhadas) e a variável global volta a estar visível.

Não é recomendado usar o mesmo nome para variáveis locais e globais. Isso pode tornar o código confuso, dificultando a leitura e a identificação de erros.

```c
int x = 5;  // variável global
void teste() {
    int x = 10;  // variável local
    printf("x = %d\n", x);
    // x = 10 (variável local tem prioridade)
}
```

9.8 Exercícios Propostos

1. O contador que não conta: O objetivo é simplesmente mostrar que alterações dentro da função não afetam a variável original quando a passagem é por valor.

Criar um programa em C com uma variável contador iniciada em 10. Escreva uma função `incrementa (int x)` que soma 5 ao valor recebido e mostre o resultado dentro da função.

No `main()`, chame a função e depois mostre o valor original de contador.

A saída a ser exibida no ecrã mostra que o valor dentro da função é o da variável local, mas no `main()` não mudou.

2. Escopo local vs. global – mesmo nome, valores diferentes: Para perceber que variáveis com o mesmo nome podem existir em escopos diferentes, declare uma variável global `numero` com valor `100`. No `main()`, declare outra variável `numero` com valor `50` e imprima-a. Crie uma função `mostraNumero()` que imprima a variável global. Por fim, mostre no ecrã: o valor local (`main()`) e o valor global (pela função).

3. Simulação de jogo – energia do jogador: Implemente um minijogo de energia. O jogador começa com 100 pontos de energia. O programa apresenta um menu:

1. Sofrer dano
2. Recuperar energia
3. Mostrar energia
4. Sair

Crie as funções:

```
int dano(int energia, int valor);
// devolve energia - valor (mínimo 0)
int curar(int energia, int valor);
// devolve energia + valor (máximo 100)
void mostrarEnergia(int energia);
// imprime a energia atual
```

Regras:

- Não aceitar valores negativos ou nulos para dano/recuperação (avisar e ignorar a operação).
- A energia nunca pode ficar abaixo de 0 nem acima de 100.
- O jogo termina quando o jogador escolher sair ou quando a energia chegar a 0 (mensagem de fim de jogo).

10 APONTADORES

Ponteiros ou apontadores são variáveis especiais que armazenam exclusivamente endereços de memória, ou seja, nunca armazenam valores diretamente.

Um apontador "aponta" para a localização de outra variável, na memória. Isso permite criar programas mais eficientes, trabalhar com estruturas mais complexas, tais como ficheiros, listas e árvores, e passar variáveis por referência para funções.

Na Figura 10-1 Variável apontador y a apontar para x, está a ser representado um bloco de memória dividido em posições numeradas, cada uma armazenando um valor. A variável `x` encontra-se no endereço fictício `16` e armazena o valor `2`. Já a variável `y`, localizada no endereço fictício `21`, contém o valor `16`. Esse valor não representa um dado comum, mas sim o endereço onde `x` está guardada. Assim, diz-se que `y` aponta para `x`. Em termos de ponteiros, `y` não guarda diretamente o valor `2`, mas sim a referência à localização de x na memória, permitindo aceder ao seu conteúdo indiretamente.

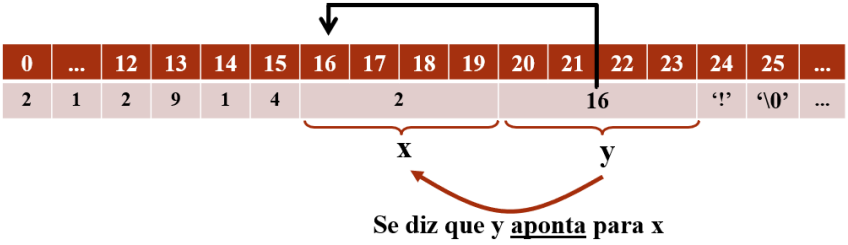
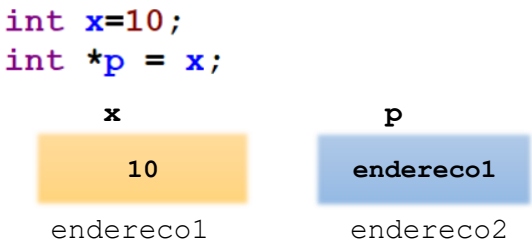


Figura 10-1 Variável apontador y a apontar para x

Um apontador deve obrigatoriamente ter o mesmo tipo da variável apontada, de forma a garantir que o compilador interpreta corretamente os dados armazenados.

Deves pensar num apontador como um "atalho" no seu computador. Numa variável normal se armazena um valor diretamente numa atribuição ou leitura, por exemplo, `int x = 10;`. Já um apontador, armazena o caminho para onde o valor está guardado, por exemplo, `int *p = &x`. O apontador não pode guardar a informação diretamente, mas sabe onde encontrar o conteúdo duma variável quando for necessário, ou seja, no endereço de memória da variável `x`, representado simbolicamente por `endereco1`.



10.1 Declaração de Apontadores

A sintaxe é simples:

```
tipo *nome_apontador;
```

A seguir, há 3 exemplos de declaração de apontadores:

```
int *ptr;      // Apontador para inteiro
float *fptr;   // Apontador para float
char *cptr;    // Apontador para char
```

Tenha atenção que o símbolo * (asterisco) na declaração é usado para indicar que é um apontador. Mas, no corpo da função, no contexto de apontadores, é também é usado para aceder o valor apontado pela variável apontador (*desreferenciar*).

A diferença entre o * de apontador e o * de multiplicação é o contexto, sendo que o primeiro é unário (atua sobre o apontador) enquanto a multiplicação é binária (exige dois valores).

Segue um exemplo de código que demonstra o uso de apontadores.

```
int x = 10;    // Variável normal
int *p = &x;  // Apontador p recebe o endereço de x
/* &x - operador devolve o endereço da variável x
   *p - operador que acessa o valor armazenado no
        endereço para o qual p aponta     */
printf("Endereço de x: %p\n", &x);
   // Mostra o endereço de x
printf("Conteúdo de x via apontador: %d\n", *p);
  // Mostra 10, o conteúdo da variável apontada
```

10.2 Operações com apontadores

Em C, é possível realizar algumas operações aritméticas e comparações diretamente com ponteiros, mas existem regras claras sobre para tal.

Entre as operações permitidas, está a operação de somar ou subtrair um valor inteiro a um apontador, deslocando-o para outra posição de memória proporcional ao tamanho do tipo de dado que ele aponta. Ou seja, estas operações deslocam o ponteiro na memória um valor proporcional aos *bytes*.

Também é válido incrementar (`pi++`) ou decrementar (`pi--`) um apontador, fazendo-o avançar ou recuar uma posição. É ainda possível subtrair dois ponteiros, obtendo como resultado um valor inteiro que representa a quantidade de elementos que os separa.

Outra operação permitida é comparar apontadores com operadores como `>`, `>=`, `<`, `<=` e `==`, verificando se apontam para a mesma posição ou para posições relativas na memória.

Por outro lado, existem operações que não são válidas. Não é permitido somar dois ponteiros (`pi + pf`), nem multiplicar ou dividir ponteiros (`pi * pf`, `pi / pf`), pois tais operações não têm significado lógico no contexto de endereços de memória.

Estas regras, exemplificadas na Tabela 23, garantem que a manipulação de ponteiros seja coerente e segura, evitando comportamentos indefinidos no programa.

Tabela 23 Exemplos de Operações com Ponteiros

Operação	**Código de Exemplo**	**Válido?**	**Explicação**
Somar um inteiro a um ponteiro	`pi + 3`	✅	Avança 3 elementos a partir da posição apontada.
Subtrair um inteiro de um ponteiro	`pi - 2`	✅	Recua 2 elementos a partir da posição apontada.
Incrementar ponteiro	`pi++`	✅	Avança 1 elemento.
Decrementar ponteiro	`pi--`	✅	Recua 1 elemento.

Operação	Código de Exemplo	Válido?	Explicação
Subtrair dois ponteiros	`pf - pi`	✅	Retorna quantos elementos existem entre pi e pf.
Comparar ponteiros	`pi > pf`	✅	Verifica qual está mais à frente na memória.
Somar dois ponteiros	`pi + pf`	❌	Não faz sentido somar endereços de memória.
Multiplicar ponteiros	`pi * pf`	❌	Não faz sentido multiplicar endereços de memória.
Dividir ponteiros	`pi / pf`	❌	Não faz sentido dividir endereços de memória.

10.3 Apontadores e arrays uni e bidimensionais

Os vetores (arrays unidimensionais) são, na prática, tratados como apontadores. O próprio nome do vetor corresponde ao endereço de memória onde está armazenado o seu primeiro elemento.

Isto significa que, ao trabalhar com arrays, pode-se utilizar a aritmética de apontadores para aceder aos elementos.

Por exemplo, a expressão `*(p + i)` é equivalente a `vetor[i]`, sendo a variável `p` um apontador para o primeiro elemento da variável `vetor`.

Este comportamento permite percorrer a memória de forma sequencial, avançando ou recuando entre elementos de acordo com o tamanho do tipo de dado armazenado.

```c
int vetor[3] = {10, 20, 30};
int *p = vetor;     // Mesmo que p = &vetor[0]

printf("%d\n", *p);    // 10 - valor na posição 0
printf("%d\n", *(p + 1));  // 20 - posição 1
printf("%d\n", *(p + 2));  // 30 - posição 2
```

10.3.1 Usar apontadores com matrizes

Uma matriz (array bidimensional) é armazenada na memória de forma sequencial, linha após linha. Assim, é possível percorrer e aceder aos seus elementos utilizando aritmética de apontadores.

A forma mais comum de aceder a um elemento é através da notação tradicional `matriz[i][j]`. No entanto, com o número de colunas e linha, também é possível utilizar a expressão: `*(p + i * colunas + j)`, onde `p` é um apontador para o primeiro elemento da matriz.

Esta abordagem, apresentada no exemplo a seguir, mostra como os apontadores permitem navegar pela memória sem depender unicamente da sintaxe com colchetes.

```c
int matriz[2][3] = {
    {1, 2, 3},
    {4, 5, 6}    };

int *p = &matriz[0][0]; // Apontador para o 1º elemento
int linhas = 2, colunas = 3;

int i = 1, j = 2;
// Elemento na linha 1, coluna 2 (valor 6)

// Acesso tradicional
printf("%d\n", matriz[i][j]);

// Acesso via apontador
printf("Via apontador = %d\n",*(p + i * colunas + j);
```

Há outra forma de usar apontadores para aceder a elementos de uma matriz bidimensional, ao percorrer a memória de forma sequencial. No exemplo a seguir, o apontador p é iniciado com o endereço do primeiro elemento da matriz (`matriz[0][0]`).

Como os elementos estão armazenados de forma contígua na memória, ao somar um valor inteiro a `p`, se desloca exatamente esse número de posições no bloco de memória.

```c
int matriz[2][3] = {{1, 2, 3}, {4, 5, 6}};
int *p = &matriz[0][0];

printf("%d\n", *(p));       // 1  - elemento [0][0]
printf("%d\n", *(p + 3));   // 4  - elemento [1][0]
printf("%d\n", *(p + 4));   // 5  - elemento [1][1]
```

10.4 Alocação Dinâmica de Memória

A linguagem C permite criar vetores cujo tamanho é conhecido em tempo de execução usando *alocação dinâmica*. Para isso, usa-se as funções da biblioteca `<stdlib.h>`, apresentadas na Tabela 24.

Tabela 24 Funções para alocação dinâmica de memória

Função	Descrição
`malloc()`	Aloca espaço na memória, não inicializado
`calloc()`	Aloca e inicializa todos os bytes com zero
`realloc()`	Redimensiona um bloco de memória já alocado anteriormente com `malloc() ou calloc()`.
`free()`	Libera o espaço alocado com `malloc/calloc/realloc`

Na sequência, são apresentadas as sintaxes correspondentes aos comandos listados na Tabela 24, que serve como referência para a sua utilização correta no código.

10.4.1 malloc()

A função `malloc()` aloca um bloco contínuo de memória com o tamanho especificado, retornando o endereço inicial desse bloco. O conteúdo da memória não é iniciado, podendo conter valores indefinidos. A sintaxe é:

```
apontador = (tipo *) malloc(n * sizeof(tipo));
```

onde `n` é o número de elementos a alocar e `sizeof(tipo)` é o número de bytes ocupados por cada elemento do tipo especificado.

Exemplo:

```
int *v;
v = (int *) malloc(5 * sizeof(int));
// espaço para 5 inteiros
```

10.4.2 calloc()

A função `calloc()` funciona de forma semelhante à `malloc()`, mas inicializa todos os bytes da memória alocada com zero. A sintaxe deste comando é:

```
apontador = (tipo *) calloc(n, sizeof(tipo));
```

onde `n` é o número de elementos a alocar e `sizeof(tipo)` é o número de bytes ocupados por cada elemento do tipo especificado.

Exemplo:

```
int *v;
v = (int *) calloc(5, sizeof(int));
// espaço para 5 inteiros, todos iniciados com 0
```

10.4.3 realloc()

A função `realloc()` ajusta o tamanho de um bloco de memória previamente alocado com `malloc()` ou `calloc()`. Pode aumentar ou diminuir o tamanho do bloco e mantém os dados existentes até ao limite do novo tamanho. A sintaxe deste comando é:

```
apontador = (tipo *) realloc(p1, n1 * sizeof(tipo));
```

onde `p1` é um apontador para o bloco de memória previamente alocado e `n1` é o novo número de elementos.

Exemplo:

```
int *v;
v = (int *) malloc(5 * sizeof(int));     // inicial
v = (int *) realloc(v, 10 * sizeof(int));
// aumenta para 10 elementos
```

A Tabela 25 mostra uma comparação simples entre as funções `malloc`, `calloc` e `realloc`.

Tabela 25 Comparação resumida das funções de alocação de memória

Função	Aloca Memória?	Inicia com 0?	Redimensiona?
`malloc`	✅	❌	❌
`calloc`	✅	✅	❌
`realloc`	❌ (usa memória já alocada)	❌	✅ (aumenta ou reduz tamanho)

Das funções apresentadas neste tópico, a `malloc()` é a mais utilizada para alocação dinâmica de memória em C. O exemplo a seguir mostra a criação de um vetor de inteiros com espaço para 10 elementos usando `malloc()`.

```c
int *vetor, i;
// Aloca espaço para 10 inteiros
vetor = (int *) malloc(10 * sizeof(int));
if (vetor == NULL) {
    printf("Erro ao alocar memória.\n");
    return 1;
}
// Preenche o vetor
for (i = 0; i < 10; i++) {
    vetor[i] = i + 1;
}
// Exibe os valores
for (i = 0; i < 10; i++) {
    printf("%d ", vetor[i]);
}
// Liberta a memória
free(vetor);
```

10.5 Vantagens e Riscos dos Apontadores

10.5.1 Vantagens

Os apontadores são um dos recursos mais poderosos da linguagem C, oferecendo diversas vantagens práticas, pois permitem aceder diretamente à memória, possibilitando manipular dados de forma mais rápida e flexível.
Em funções que precisam trabalhar com estruturas de dados grandes, como matrizes ou registos complexos, o uso de apontadores torna o processamento mais eficiente, pois evita cópias desnecessárias.
Além disso, são essenciais para a alocação dinâmica de memória, recurso que permite criar e gerir variáveis em tempo de execução, adaptando o programa às necessidades do momento.
Outra vantagem importante é que apontadores possibilitam a passagem de parâmetros por referência (tema abordado no Capítulo 11), permitindo que uma função altere diretamente o conteúdo de variáveis declaradas fora dela.
Por fim, apontadores são fundamentais na implementação de estruturas de dados dinâmicas, como listas ligadas, árvores e buffers, que dependem dessa flexibilidade para gerir a memória de forma eficiente.

10.5.2 Riscos no Uso de Apontadores

O uso incorreto de apontadores pode causar erros graves e difíceis de identificar. Alguns dos problemas mais comuns incluem:

- Apontar para zonas inválidas da memória: Usar apontadores com endereços incorretos pode levar a comportamentos imprevisíveis.

- Esquecer de iniciar: Um apontador não iniciado não aponta para nenhum sítio válido. Pode conter lixo de memória.
- Liberar memória errada (alocação dinâmica): Libertar um endereço que não foi alocado corretamente ou liberar duas vezes pode causar falhas.
- Acesso a apontadores após libertar memória: Mesmo após chamar `free()`, o apontador ainda guarda o endereço. É recomendável atribuir NULL após liberar.

Uma boa prática é sempre iniciar os apontadores com NULL. Assim, é possível verificar com segurança se o apontador está pronto para ser usado.

10.6 Resumo das Operações com Apontadores

Ao trabalhar com apontadores em C, dois símbolos são fundamentais: o * (asterisco) e o & (e comercial). O primeiro serve para aceder o valor apontado por um apontador e o segundo para obter o endereço de memória de uma variável.

A Tabela 26 apresenta um resumo simples e direto dessas operações, com exemplos e a interpretação de cada uma.

Tabela 26 Resumos das operações fundamentais com ponteiros

Símbolo	Significado	Exemplo	Leitura
`*`	Desreferência (valor do endereço)	`*p`	Valor armazenado no apontador
`&`	Endereço de uma variável	`&x`	Onde está armazenado x

10.7 Exercícios Propostos

1. Soma com apontadores: Crie um programa para ler dois números inteiros introduzidos pelo utilizador, armazene-os em variáveis distintas e utilize apontadores para calcular a soma. O programa deve exibir: os dois números introduzidos, bem como o resultado da soma calculada através dos apontadores.

2. Inversão de vetor com apontadores: Criar um vetor de tamanho fixo (por exemplo, 6 posições) e preencha-o com valores introduzidos pelo utilizador. Em seguida, utilizando apenas aritmética de apontadores dentro do próprio `main()`, inverta a ordem dos elementos, armazenando-os num segundo vetor. O programa deve exibir o vetor original e o vetor invertido.

3. Alocação Dinâmica com malloc(): Solicite ao utilizador o tamanho de um vetor, aloque dinamicamente a memória usando `malloc()` e preencha-o com valores introduzidos pelo teclado. Depois, calcule e exiba: o maior valor, o menor valor e a média dos elementos.

4. Redimensionamento com realloc(): Aloque dinamicamente um vetor com 5 posições e preencha-o com valores introduzidos pelo utilizador. Use `realloc()` para expandir o vetor para 10 posições, preencha as novas posições e exiba todos os elementos do vetor após a expansão. Para além, exiba a soma total dos elementos.

5. Inicialização com calloc(): Crie um vetor de 100 inteiros usando `calloc()` e confirme que todos estão inicializados com zero. Depois, preencha-o com valores de 1 a 100 e exiba os 10 primeiros valores do vetor e a soma total dos elementos.

11 PASSAGEM DE PARÂMETROS POR REFERÊNCIA

Todas as funções apresentadas até o momento, se utilizaram apenas da passagem de parâmetro por valor. Ou seja, a função recebe uma cópia da variável e qualquer alteração que venha a ser feita dentro da função não afeta a variável original.

Na passagem de parâmetro por referência, em vez de copiar o valor, passa-se o endereço da variável original. A função pode então aceder diretamente à memória dessa variável e modificar o seu conteúdo.

Essa técnica é útil quando:

- Se quer que a função altere os valores originais
- Se necessita retornar vários valores de uma só vez
- Há estruturas grandes (eficiência).

11.1 Como se usa passagem de parâmetro por referência

Para passar uma variável por referência em C, usa-se apontadores.

Em vez de passar `int x`, passa-se `int *x` e se utiliza `*x` dentro da função para aceder/modificar o valor.

Um exemplo de troca de conteúdo de variáveis está apresentado logo a seguir.

```c
void trocar(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 8;
    trocar(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    // x = 8, y = 5
    return 0;
}
```

Note o uso de `&x` e `&y` ao chamar a função. Isso envia os endereços das variáveis. A função `trocar()` utiliza apontadores para modificar os valores originais.

11.2 Diferença em relação à Passagem de parâmetro por valor

Em C, como já visto, quando se chama uma função, pode-se passar os dados de duas formas: por valor ou por referência. A diferença está no que é realmente enviado para a função e no que ela pode (ou não) alterar.

Na passagem de parâmetro por valor, a função recebe apenas uma cópia do valor da variável. Isso significa que, mesmo que a função altere esse valor internamente, a variável original permanece inalterada fora da função.

Já na passagem por referência, o que é enviado para a função é o endereço de memória da variável (através de um apontador). Com isso, qualquer alteração feita dentro da função afeta diretamente a variável original.

A Tabela 27 apresenta um resumo comparativo entre essas duas formas de passagem de parâmetros:

Tabela 27 Comparação entre as passagens de parâmetro em C

Tipo	O que é enviado	Pode alterar a original?
Por Valor	Cópia do valor	Não
Por Referência	Endereço (apontador)	Sim

Após compreender a diferença conceitual, a seguir se apresenta um exemplo prático comparativo.

A. **Por Valor** – não altera a variável original fora da função:

```
void dobrar(int n) {
    n = n * 2;
}
```

B. **Por Referência** – recebe o endereço da variável e altera o valor original:

```
void dobrar(int *n) {
    *n = *n * 2;
}
```

C. Uso no main() – com passagem por referência

```
int valor = 10;
dobrar(&valor);
printf("%d\n", valor);  // 20
```

11.3 Vetores como Parâmetros

Em C, ao passar um vetor como parâmetro para uma função se está a enviar o endereço do seu primeiro elemento. Na prática, isso significa que o vetor é passado por referência e a função pode modificar diretamente os valores do vetor original.

Embora a sintaxe seja semelhante à de uma passagem de parâmetro por valor, o comportamento é diferente: não existe uma cópia completa do vetor. Isso acontece porque em C, não é possível passar automaticamente uma sequência de valores individuais (como um vetor completo) por valor, a menos que se declare explicitamente uma variável para cada elemento. O que é uma ideia nada prática e, muitas vezes, impossível de realizar.

Assim, a passagem de vetores a funções em C é sempre feita por referência, permitindo que qualquer alteração nos elementos dentro da função afete o vetor original no programa principal.

```c
void preencher(int v[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        v[i] = i * 2;
    }
}

int main() {
    int numeros[5];
    preencher(numeros, 5);

    for (int i = 0; i < 5; i++) {
        printf("%d ", numeros[i]);
    }
    return 0;
```

Neste exemplo, a função `preencher()` modifica diretamente os valores do vetor `numeros`. O vetor está a ser preenchido e possui, ao final, os valores 0, 2, 4, 6, 8, nas suas respetivas posições.

11.4 Matrizes como Parâmetros

Em C, ao passar uma matriz (array bidimensional) como parâmetro para uma função, é obrigatório indicar o número de colunas da matriz (ou seja, a segunda dimensão). Isso acontece porque, ao contrário da primeira dimensão (linhas), que pode ser omitida, o compilador precisa saber quantos elementos há por linha para calcular corretamente os endereços de memória durante o acesso.

```c
void mostrarMatriz(int m[][3], int linhas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", m[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int matriz[2][3] = {{1, 2, 3}, {4, 5, 6}};
    mostrarMatriz(matriz, 2);
    return 0;
}
```

11.5 Retorno de Vetores

Em C, não é possível devolver diretamente um vetor sítio declarado dentro de uma função, como por exemplo `int v[10]`, porque esse vetor é alocado na memória automática, também chamada de *stack* (pilha). A memória da *stack* é automaticamente libertada assim que a função termina, o que significa que qualquer vetor sítio deixa de existir e o seu conteúdo torna-se inválido após o fim da função.

Mesmo que se utilize a instrução `return v;` antes da última linha da função, o que está a ser devolvido é o endereço de uma zona de memória temporária, que será libertada automaticamente ao terminar a função. Esse endereço pode continuar a apontar para valores aparentes durante algum tempo (por exemplo em testes simples), mas essa memória pode ser reutilizada por outras funções ou operações futuras. O resultado disso é um comportamento indefinido, que pode provocar erros imprevisíveis ou dados corrompidos.

Além disso, vetores contêm vários elementos e, por isso, não podem ser devolvidos como uma variável simples (como `int` ou `float`). Em C, as funções podem retornar apenas um valor por vez (esse conceito é muito importante para entender as funções em C) e esse valor precisa ter tipo simples ou ser um apontador.

A solução correta para retornar vetores é usar alocação dinâmica de memória com `malloc()`, que reserva espaço na *heap*, uma área de memória que não é libertada automaticamente ao fim da função. Dessa forma, a função pode retornar com segurança um apontador para o primeiro elemento do vetor e

esse apontador permanecerá válido até que o programador decida libertar explicitamente essa memória com a função `free()`.

No exemplo a seguir, a função `criarVetor()` recebe um tamanho e aloca dinamicamente memória suficiente para armazenar um vetor de inteiros com essa dimensão. O espaço é reservado na *heap*, utilizando `malloc()` e a função retorna um apontador para o primeiro elemento do vetor.

Diferente de um vetor local (alocado na *stack*), o vetor criado com `malloc()` permanece válido mesmo após o fim da função, porque a memória da *heap* não é libertada automaticamente. No entanto essa abordagem exige cuidado: é responsabilidade do programador libertar a memória alocada com a função `free()` assim que o vetor deixar de ser necessário. Caso contrário, haverá um vazamento de memória (*memory leak*), o que pode comprometer o desempenho do programa em execuções prolongadas ou com muitos dados.

```
int* criarVetor(int tamanho) {
    int* v = (int*) malloc(tamanho * sizeof(int));
    for (int i = 0; i < tamanho; i++) {
        v[i] = i;
    }
    return v;
}
```

11.6 Exercícios Propostos

1. Transformação de Poder: Num jogo de combate, dois personagens começam com energia cheia. Crie uma função que receba, por referência, os pontos de energia dos dois e reduza-os à metade para simular o efeito de um ataque especial.
Dica*:* use divisão inteira para simplificar o cálculo.

2. Batalha de Números: Dois jogadores inserem os seus números secretos. Faça uma função que receba esses números por valor e devolva, por referência, qual é o maior e qual é o menor, determinando o "campeão" e o "desafiante".

3. Loot Aleatório: Num jogo, um baú contém vários itens aleatórios. Escreva uma função que receba, por referência, um vetor vazio e o seu tamanho, preenchendo-o com valores aleatórios entre 0 e 100 representando os pontos de cada item.
Dica*:* use `rand()` e `srand(time(NULL))` para gerar números diferentes a cada execução.

12 STRUCTS: DEFININDO TIPOS COMPOSTOS

A linguagem C permite que o programador crie seus próprios tipos de dados por meio de uma construção chamada `struct`, abreviação de *structure* (estrutura). Uma `struct` agrupa, sob um único nome, diferentes variáveis (chamadas campos ou membros), possivelmente de tipos distintos, que representam de forma mais fiel entidades do mundo real. E este tipo de dado é criado pelo próprio programador.

Essa capacidade de agrupar dados heterogêneos é essencial para a construção de programas organizados, modulares e que espelhem melhor os conceitos do domínio da aplicação.

É de extrema importância perceber que a ordem dos campos declarados em uma `struct` deve ser cuidadosamente planeada e respeitada. Em C, os campos são armazenados e acedidos na memória na ordem em que foram definidos. Além disso, é comum iniciar a definição da `struct` com um campo que identifique unicamente o registo, como por exemplo um número de identificação (id) ou um código (codigo). Esse campo serve como uma chave primária, facilitando buscas, comparações e organização dos dados em estruturas mais complexas, como vetores ou listas.

12.1 Por que usar structs

Considere o caso em que é necessário representar informações sobre uma pessoa: nome, idade e altura. Se utilizar apenas variáveis simples, é necessário declarar separadamente:

```
char nome[30];
int idade;
float altura;
```

Mas isso rapidamente se torna confuso e difícil de gerenciar ao lidar com vários registos. Com `structs`, se cria um tipo de dado composto, mais expressivo e organizado.

```
typedef struct {
    char nome[30];
    int idade;
    float altura;
} Pessoa;
```

Agora, `Pessoa` é um tipo de dado definido pelo programador e é possível declarar variáveis do tipo Pessoa da mesma forma que se faz com `int` ou `float`.

12.2 Definir e aceder a structs

A forma mais básica de definir uma `struct` em C é a seguinte:

```
struct Pessoa {
    char nome[30];
    int idade;
    float altura;
};
```

Essa definição cria um novo tipo chamado `struct Pessoa`. É importante frisar que `struct Pessoa` é o tipo do dado e não o nome da variável.

Para declarar variáveis desse tipo, usa-se a palavra-chave `struct` novamente:

```
struct Pessoa aluno;
```

Ou seja, `aluno` é uma variável do tipo `struct Pessoa`.

A atribuição de valores aos campos da `struct` é feita usando o operador ponto (`.`) para uma estrutura que não seja do tipo apontador. Esta utiliza o operador seta (`->`), que é visto a seguir.

```
strcpy(aluno.nome, "João");
aluno.idade = 20;
aluno.altura = 1.75;
```

Nota importante: Como nome é um vetor de caracteres (uma string em C), é necessário utilizar a função `strcpy()` da biblioteca `<string.h>` para copiar o texto.

Exemplo

```
struct Pessoa {
    char nome[30];
    int idade;
    float altura;
};
//definição da struct e NÃO declaração da variável
int main() {
    struct Pessoa aluno; //variável local declarada
    strcpy(aluno.nome, "João");
    aluno.idade = 20;
    aluno.altura = 1.75;
    printf("Nome: %s\n", aluno.nome);
    printf("Idade: %d\n", aluno.idade);
    printf("Altura: %.2f\n", aluno.altura);
    return 0;
}
```

Esse programa declara uma `struct` chamada `Pessoa`, cria uma variável chamada `aluno` e imprime seus dados.

Uma observação importante é que em C, é possível declarar uma variável ao mesmo tempo em que se define a `struct`. Isso pode ser feito diretamente dentro de uma função, como no exemplo a seguir:

```
int main() {
    struct Pessoa {
        char nome[30];
        int idade;
        float altura;
    } aluno;
    // Uso da variável aluno
    strcpy(aluno.nome, "João");
    aluno.idade = 20;
    aluno.altura = 1.75;
...
}
```

Neste caso, a estrutura `Pessoa` e a variável `aluno` estão sendo definidas localmente dentro da função `main`.

Cuidado: essa técnica limita o uso da `struct` ao escopo da função onde ela foi definida. Isso significa que outras funções do programa não poderão usar o tipo `struct Pessoa`, porque ele não estará visível fora da função `main`. Essa prática deve ser evitada quando o tipo criado precisa ser reutilizado em diferentes partes do programa, como por exemplo em várias funções em vetores globais ou em parâmetros de funções.

É muito recomendado que se defina a `struct` fora de qualquer função, geralmente no início do programa ou num ficheiro de cabeçalho (`.h`). Dessa forma ela estará acessível para todo o código e poderá ser reutilizada com segurança.

12.3 Simplificar com typedef

Em programas maiores, escrever `struct` seguido do nome do tipo sempre que se declara uma variável pode tornar o código repetitivo e menos legível. Para simplificar, a linguagem C oferece o recurso `typedef` para criar um nome alternativo (um apelido) para o tipo `struct`.

Veja a seguir como reescrever a definição anterior utilizando `typedef`:

```
typedef struct {
    char nome[30];
    int idade;
    float altura;
} Pessoa;
```

Agora, pode-se declarar variáveis diretamente, apenas com:

```
Pessoa aluno;
```

12.4 Vetores de Structs

Também é possível criar vetores de `structs` ou seja, armazenar vários registos do mesmo tipo. Isso é útil, por exemplo, para representar uma turma com vários alunos. A sintaxe é semelhante à de vetores comuns:

```
struct Pessoa turma[50];
```

Atribui-se valores a cada elemento do vetor usando o índice e o operador ponto (**.**).

No exemplo a seguir, o programa realiza o cadastro de 50 alunos, utilizando `structs`, vetores, funções e passagem de parâmetros.

Por ser um vetor, a passagem de parâmetro é por referência na leitura (para alterar diretamente o conteúdo) e por valor na impressão (quando não há necessidade de modificar os dados).

A leitura dos nomes é feita com a função `fgets()`, que permite capturar nomes com espaços (exemplo: "João da Silva"), ao contrário de `scanf("%s", ...)`, que interrompe a leitura ao encontrar o primeiro espaço em branco.

Como `fgets()` é utilizada após `scanf()` (para idade e altura), é necessário limpar corretamente o carácter de nova linha `(\n)` deixado no buffer de entrada. Para isso, utiliza-se a função `getchar()`, conforme explicado no capítulo 8 deste livro.

```c
#define TAM 50  // número máximo de alunos
    struct Pessoa {
        char nome[30];
        int idade;
        float altura;
    };

    // Função para ler os dados de um aluno
    void lerPessoa(struct Pessoa *p) {
        getchar(); // limpar o buffer do teclado
        fgets(p->nome, sizeof(p->nome), stdin);
        p->nome[strcspn(p->nome, "\n")] = '\0';
        // remove o '\n' e substitui por '\0'
        printf("Idade: ");
        scanf("%d", &p->idade);
        printf("Altura (em metros): ");
        scanf("%f", &p->altura);
    }
```

```c
// Função para imprimir os dados de um aluno
void imprimirPessoa(struct Pessoa p) {
    printf("Nome: %s\n", p.nome);
    printf("Idade: %d anos\n", p.idade);
    printf("Altura: %.2f m\n", p.altura);
}
int main() {
    struct Pessoa turma[TAM];
    printf("Cadastro de %d alunos\n\n", TAM);
    for (int i = 0; i < TAM; i++) {
        printf("Nome do aluno %d: ", i + 1);
        lerPessoa(&turma[i]);
        printf("\n");
    }
    printf("Dados cadastrados:\n\n");
    for (int i = 0; i < TAM; i++) {
        printf("Aluno %d:\n", i + 1);
        imprimirPessoa(turma[i]);
        printf("\n");
    }
    return 0;
}
```

12.5 Apontadores para struct

Assim como outros tipos em C, também é possível utilizar apontadores para structs. Ao trabalhar com apontadores, os campos da `struct` são acedidos com o operador seta (`->`) e não com o ponto (`.`).

Para associar um apontador a uma `struct`, usa-se o operador de endereço (`&`) sobre uma variável do tipo apontador.

```c
struct Pessoa aluno;
struct Pessoa *ptr = &aluno;
```

Atribui-se valores da seguinte forma:

```
strcpy(ptr->nome, "João");
ptr->idade = 20;
ptr->altura = 1.75;
```

Exemplo:

```
#include <stdio.h>
#include <string.h>

struct Pessoa {
    char nome[30];
    int idade;
    float altura;
};

int main() {
    struct Pessoa aluno;
    struct Pessoa *ptr = &aluno;
    strcpy(ptr->nome, "João");
    ptr->idade = 20;
    ptr->altura = 1.75;
    printf("Nome: %s\n", ptr->nome);
    printf("Idade: %d\n", ptr->idade);
    printf("Altura: %.2f\n", ptr->altura);
    return 0;
}
```

O uso de apontadores para structs é essencial quando se deseja passar structs para funções por referência ou quando se trabalha com alocação dinâmica.

12.6 Structs aninhadas

Em alguns casos, é necessário representar dados mais complexos e hierárquicos, onde um dos campos de uma `struct` também é uma `struct`. Isso é chamado de `struct` aninhada (ou estrutura dentro de estrutura).

Essa técnica permite construir tipos compostos mais ricos e modelar entidades mais completas com clareza e organização.

Exemplo: pessoa com data de nascimento

Considere que se quer incluir a data de nascimento de uma pessoa. Pode-se criar uma `struct Data` e usá-la como campo dentro da `struct`

```
struct Data {
    int dia;
    int mes;
    int ano;
};

struct Pessoa {
    char nome[30];
    int idade;
    float altura;
    struct Data nascimento;
};
```

Agora, além de `nome, idade e altura, a struct Pessoa` também possui um campo `nascimento`, que contém uma `struct Data` com `dia, mes e ano`.

Vale destacar que uma `struct` aninhada pode ser reutilizada em diferentes campos da `struct` principal. Por exemplo, uma pessoa pode ter tanto a data

de nascimento quanto a data de cadastro no sistema. Nesse caso, a `struct Pessoa` poderia conter dois campos do tipo `struct Data`, como mostrado a seguir:

```
struct Data nascimento, cadastro;
```

O acesso aos campos internos é feito utilizando o operador ponto `(.)` ou `(->)`, conforme o caso, em sequência:

```
struct Pessoa aluno;

strcpy(aluno.nome, "Maria Sousa");
aluno.idade = 20;
aluno.altura = 1.65;
aluno.nascimento.dia = 1;
aluno.nascimento.mes = 1;
aluno.nascimento.ano = 2005;
```

Impressão dos dados:

```
printf("Data de nascimento: %02d/%02d/%04d\n",
    aluno.nascimento.dia, aluno.nascimento.mes,
    aluno.nascimento.ano);
```

Esse tipo de organização é particularmente útil em programas que manipulam dados complexos, como sistemas de cadastro, bases de dados, jogos ou aplicações gráficas.

Structs aninhadas também podem ser combinadas com `typedef`, facilitando ainda mais a escrita e leitura do código.

Na função `printf`, as sequências como `%02d` ou `%04d` indicam formatação com largura mínima e preenchimento. No exemplo acima, `%02d` significa que o número será impresso com, no mínimo, dois dígitos. Se tiver apenas um dígito (como o número 1), ele será exibido como 01. Já `%04d` imprime o número com, no mínimo, quatro dígitos (por exemplo, o ano 2025).
Esse tipo de formatação é útil para manter a saída visualmente alinhada, especialmente em datas, valores monetários ou tabelas.

Exemplo completo de um programa de cadastro de pessoas com suas respetivas idades e endereços. Nesse exemplo a `struct Endereco` é definida primeiro e depois usada dentro da `struct Pessoa`.
Para aceder campos aninhados, usar a notação, por exemplo, `pessoa.endereco.cp`. Isso demonstra como organizar dados hierárquicos (exemplo: pessoa → endereço → CP).

```c
#include <stdio.h>
#include <string.h>

// Struct aninhada para morada
typedef struct {
    char rua[50];
    int numero;
    char cidade[30];
    char cp[9]; // Formato: "1234-000"
} Morada;
// Struct principal que contém a struct Endereco
typedef struct {
    char nome[50];
    int idade;
    Morada endereco; // Campo do tipo Morada
} Pessoa;

int main() {
    Pessoa pessoa;

    // Preencher os dados
    strcpy(pessoa.nome, "Marta Oliveira");
    pessoa.idade = 21;

    // Acessando campos da struct aninhada
    strcpy(pessoa.endereco.rua, "Rua das Flores");
    pessoa.endereco.numero = 100;
    strcpy(pessoa.endereco.cidade, "Lisboa");
    strcpy(pessoa.endereco.cp, "1700-123");

    // Exibindo os dados
    printf("Nome: %s\n", pessoa.nome);
    printf("Idade: %d\n", pessoa.idade);
    printf("Morada: %s, %d - %s\n", pessoa.endereco.rua,
           pessoa.endereco.numero, pessoa.endereco.cidade);
    printf("CP: %s\n", pessoa.endereco.cp);
    return 0;
}
```

Saída

```
Nome: Marta Oliveira
Idade: 21
Morada: Rua das Flores, 100 - Lisboa
CP: 1700-123
```

12.7 Exercícios Propostos

1. Cadastro de crianças, alocação dinâmica de memória: Defina, nesta ordem, uma `struct Pessoa` com os campos: `nome` (até 50 caracteres), `idade` (inteiro, entre 5 e 12 anos), `altura` (`float`, entre 1.0 m e 1.5 m). O programa deve:

1. Perguntar quantas crianças serão registadas;
2. Alocar memória dinamicamente para armazenar os registos;
3. Ler os dados de cada criança com **validação**, usando uma função `lerPessoa()` (passagem por referência);
4. Exibir os registos com uma função `imprimirPessoa()` (passagem por valor);
5. Libertar a memória no final.

Dicas:

- Use `fgets()` para ler o `nome`, `removerNovaLinha()` para retirar o `\n;`
- Valide idade e altura com ciclos `do...while` até que os valores estejam no intervalo correto.

2. Continuação do exercício 1: A partir do programa desenvolvido no exercício 1, na alínea anterior, acrescente novas funcionalidades para torná-lo mais completo e interativo. O programa deve manter a `struct Pessoa` e as funções já criadas (`lerPessoa()` e `imprimirPessoa()`). Adicionar novas funções para:

- Listar todos os registos
- Calcular e mostrar a média das idades
- Identificar e mostrar a criança mais alta
- Procurar criança por `nome` (ignorar maiúsculas/minúsculas)
- Contar quantas crianças têm altura acima da média
- Ordenar a lista por altura (ascendente)
- Alterar os dados de uma criança pelo índice
- Implementar um `menu` que repita até o utilizador escolher "Sair".

Requisitos técnicos:

- Passagem por valor: funções que apenas consultam dados (`imprimirPessoa()`, `mediaIdades()`, `maiorAltura()`)
- Passagem por referência: funções que modificam dados (`alterarPessoa()`, `ordenarAltura()`, `lerPessoa()`)
- Pesquisa: normalizar para minúsculas ou maiúsculas antes de comparar.

Dica: Use funções curtas e bem definidas para cada tarefa. Isso torna o código mais claro e evita repetições.

```
Quantidade de alunos/criancas: 2

--- Registo 1 ---
Nome: Maria Joao Costa
Idade (5 a 12): 4
Valor invalido.
Idade (5 a 12): 6
Altura (1.0 a 1.5 m): 1.35

--- Registo 2 ---
Nome: Goncalo Bastos
Idade (5 a 12): 9
Altura (1.0 a 1.5 m): 1

===== Menu 12.1 =====
1) Listar todos
2) Media das idades
3) Pessoa mais alta
4) Procurar por nome
5) Contar acima da media de altura
6) Ordenar por altura (asc)
7) Alterar registo por indice
8) Sair
Opcao: 1

--- Lista ---
Nome: Maria Joao Costa     | Idade:  6 | Altura: 1.35 m
Nome: Goncalo Bastos       | Idade:  9 | Altura: 1.00 m
```

3. Campeonato de Corridas: Crie um programa para gerir um campeonato de corridas. Defina:

- Uma `struct Data` com os campos `dia, mes e ano`.
- Uma `struct Corrida` com os campos: `piloto` (até 50 caracteres), `pista` (até 50 caracteres), `melhorVolta` (`float`, tempo em segundos), `dataProva` (do tipo `Data`)

O programa deve perguntar quantas corridas serão registadas e alocar memória dinamicamente para armazenar os registos.

Implementar as seguintes funções:

- `lerCorrida()` – recebe um apontador para `Corrida` e lê os dados, a validar que `melhorVolta` é positiva
- `imprimirCorrida()` – recebe uma `Corrida` e exibe os dados
- `melhorTempo()` – recebe o vetor de corridas e retorna a corrida com o menor tempo de volta
- `alterarTempo()` – recebe o vetor e permite alterar o tempo de volta de um piloto específico.

Criar um `menu` interativo que permita:

- Inserir os dados de todas as corridas.
- Listar todas as corridas.
- Mostrar a corrida mais rápida.
- Alterar o tempo de um piloto.
- Sair do programa.

Libertar a memória no final.

13 FICHEIROS EM C

Na linguagem C, é possível ler e gravar dados em ficheiros, tanto no formato de texto quanto binário. Isso permite que os dados de um programa sejam armazenados em unidades permanentes de armazenamento, como HDs, SSDs, *pendrives* ou outros dispositivos, podendo ser reutilizados posteriormente.

Para manipular ficheiros em C, utiliza-se um apontador especial do tipo `FILE`, declarado com a seguinte sintaxe:

```
FILE *nomeApontador;
```

Esse apontador representa um fluxo de dados associado a um ficheiro no disco. Todas as operações de leitura ou escrita passam por esse apontador `FILE` que é controlado internamente pela linguagem C, que mantém informações como a posição atual de leitura ou gravação no ficheiro. Sem esse apontador, não é possível interagir com o conteúdo armazenado no ficheiro.

Após concluir as operações de escrita, leitura ou qualquer outro tipo de manipulação com o ficheiro, é fundamental utilizar a função `fclose(nomeApontador);`.

Essa chamada encerra o fluxo de dados associado ao ficheiro, garantindo que todas as informações pendentes sejam gravadas corretamente no disco e que os recursos internos do sistema sejam liberados de forma segura. Ignorar esse passo pode resultar em perda de dados ou comportamentos inesperados.

A Tabela 28 apresenta uma comparação direta entre ficheiros de texto e ficheiros binários, destacando as principais vantagens, desvantagens e exemplos de utilização de cada formato. Este tipo de comparação ajuda na escolha do formato mais adequado para um determinado projeto, considerando fatores como legibilidade, desempenho e portabilidade.

Tabela 28 Comparação entre ficheiros texto e binário

Critério	**Ficheiros de Texto**	**Ficheiros Binários**
Formato	Legível por humanos (ASCII/UTF-8)	Dados brutos (bytes)
Tamanho	Geralmente maior – por exemplo, o número 12345 ocupa **5 bytes** como texto	Geralmente menor – por exemplo, um `int` com valor 12345 ocupa **4 bytes** em binário
Velocidade	Leitura/escrita mais lenta, devido à conversão para texto	Leitura/escrita mais rápida, pois trabalha diretamente com dados
Edição manual	Pode ser aberto e alterado em editores simples	Não legível sem programa adequado
Portabilidade	Alta – independente da arquitetura e *endianness*	Pode haver incompatibilidades entre sistemas com *endianness* diferente
Uso típico	Configurações (`.ini`, `.json`), registos de log	Armazenamento de dados estruturados (structs, vetores), imagens, áudios, bases de dados

Ficheiros texto são mais indicados quando é necessário que os dados possam ser lidos ou editados por humanos, como em ficheiros CSV, JSON ou para configurações. Também são preferíveis em situações onde a portabilidade entre diferentes sistemas operativos é essencial, já que seu formato é independente da arquitetura da máquina.

Por outro lado, ficheiros binários são ideais quando a performance e o tamanho do ficheiro são fatores importantes, como em jogos, bancos de dados ou aplicações que usam grandes volumes de informação. Estes ficheiros também são a melhor escolha para salvar estruturas (`structs`) ou dados complexos diretamente da memória, sem necessidade de conversão para texto, garantindo mais precisão e eficiência.

13.1 Ficheiros de Texto

Ficheiros de texto armazenam dados em formato legível por humanos, utilizando letras, números e símbolos representados como caracteres. São compostos por sequências de texto, como as que se vê num editor comum (por exemplo, Bloco de Notas) e podem ser abertos e lidos facilmente sem programas especiais.

13.1.1 Abertura de Ficheiros

Antes de ler ou gravar dados, é necessário abrir o ficheiro com a função `fopen()`. Essa função retorna um apontador para o 1º byte do ficheiro. Caso a abertura falhe (por exemplo, se o ficheiro não existir no modo leitura), o apontador retornado será NULL.

```c
FILE *fp = fopen("dados.txt", "r");
// Modos: "r", "w", "a", conforme Tabela 29
// Pode também: "rt", "wt", "at", onde t de texto
if (fp == NULL) {
    perror("Erro ao abrir ficheiro");
    exit(1);
```

Ao tentar abrir um ficheiro com `fopen()`, é essencial verificar se a operação foi bem-sucedida. Caso o apontador retornado seja `NULL`, significa que o ficheiro não pôde ser aberto (por exemplo, por inexistência, falta de permissões ou caminho incorreto). Para diagnosticar o motivo da falha, pode-se usar a função `perror()`.

A função `perror()` exibe uma mensagem de erro baseada no último erro ocorrido com operações de entrada e saída. Ela recebe como argumento uma string personalizada que será impressa antes da descrição do erro do sistema.

A Tabela 29 apresenta os principais modos de abertura para ficheiros de texto. O modificador t, que indica modo texto, é opcional e, por isso, não está explícito na tabela.

Tabela 29 Modos de abertura de ficheiros texto

Modo	Significado
`"r"`	Abre para leitura
`"w"`	Abre para escrita (apaga o conteúdo anterior)
`"a"`	Abre para adicionar ao final
`"r+"`	Leitura e escrita
`"w+"`	Escrita e leitura (apaga o conteúdo)
`"a+"`	Leitura e escrita (adiciona ao final)

13.1.2 Escrita no Ficheiro Texto

Após abrir um ficheiro no modo adequado para escrita (por exemplo, `"w"` ou `"a"`), é possível registar informações dentro dele utilizando funções específicas da biblioteca padrão da linguagem C. Essas funções permitem escrever dados formatados ou caracteres individuais, controlando o conteúdo que será armazenado.

A escrita em ficheiros texto é semelhante à impressão no ecrã com `printf()`, porém, em vez de enviar os dados ao terminal, eles são enviados ao ficheiro através de um apontador do tipo `FILE*`.
As principais funções para escrever em ficheiros são `fprintf()`, `fputs()` e `fputc()`, cada qual com uma finalidade específica.

a) fprintf()

A função `fprintf()` é a mais completa entre elas. Sua sintaxe é semelhante à da função `printf()`, com a diferença de que o primeiro argumento deve ser o apontador para o ficheiro.

```
FILE *fp;
...
fprintf(fp, "expressaoControlo", listaVariaveis);
```

Os dados serão gravados no ficheiro referenciado por `fp`, com as formatações conhecidas de `printf()`, como `%s`, `%d`, `%f`, `%c`, etc., na `expressaoControlo`.

b) `fputs()`

A função `fputs` serve para gravar diretamente uma string (sem formatação) no ficheiro. Ela não adiciona automaticamente o carácter de nova linha `(\n)`, então, se necessário, esse carácter deve ser incluído manualmente.

```
fputs("Registo iniciado\n", fp);
```

onde `fp` é o apontador para o ficheiro.
É uma opção útil quando se deseja gravar texto fixo ou já formatado, sem variáveis.

c) `fputc()`

A função `fputc()` grava um único carácter por vez no ficheiro.

```
fputc('A', fp);
```

Pode ser usada em conjunto com laços para escrever carácter por carácter, ou para inserir símbolos específicos no texto.

Cada uma dessas funções tem seu uso ideal. Em geral, `fprintf()` é a mais usada quando se deseja gravar dados variáveis e formatados, enquanto `fputs()` e `fputc()` são úteis para operações mais simples ou mais controladas.

13.1.3 Leitura de Ficheiro Texto

Após abrir um ficheiro no modo apropriado para leitura (como por exemplo, `"r"` ou `"r+"`), é possível ler seu conteúdo utilizando funções específicas da linguagem C. Essas funções permitem capturar dados formatados, linhas completas ou caracteres individuais, conforme a necessidade do programa.

A leitura de ficheiros texto em C é semelhante à leitura feita a partir do teclado, mas em vez de vir da entrada padrão (`stdin`), os dados são lidos do ficheiro por meio de um apontador do tipo `FILE*`.

As principais funções utilizadas para ler os dados que são obtidos de ficheiros são `fscanf()`,`fgets()`, `fgetc()` e `sscanf()`, cada uma com características e aplicações específicas, conforme apresentado a seguir.

a) `fscanf()`

A função `fscanf()` funciona de maneira parecida com `scanf()`, lendo dados formatados a partir de um ficheiro. O primeiro argumento é o apontador para o ficheiro, seguido da expressão de controlo e da lista de variáveis:

```
FILE *fp;
...
fscanf(fp, "expressaoControlo", &var1, &var2, ...);
```

Esta função é útil quando os dados do ficheiro seguem um padrão fixo. No entanto, deve-se ter atenção ao tipo e à ordem dos dados, pois qualquer divergência pode causar falhas na leitura.

Assim como ocorre com a função `scanf()`, a função `fscanf()` interrompe a leitura de strings no primeiro espaço em branco encontrado. Isso significa que, ao tentar ler nomes compostos, como "João da Silva", apenas a palavra "João" será armazenada e o restante será ignorado. Por isso, para ler textos com espaços, recomenda-se utilizar a função `fgets()`, que é capaz de capturar toda a linha, incluindo os espaços entre palavras.

b) fgets()

A função `fgets()` é utilizada para ler uma linha completa do ficheiro, incluindo espaços em branco. Está função lê no máximo `n-1` caracteres ou até encontrar um carácter de nova linha (`\n`) ou fim do ficheiro. A sintaxe é:

```
fgets(string, tamanho, fp);
```

O resultado é armazenado como uma string, com o `\n` incluído, caso haja espaço suficiente. Por isso, muitas vezes é necessário remover esse carácter após a leitura, utilizando, por exemplo este comando:

```
string[strcspn(string, "\n")] = '\0';
```

c) fgetc()

A função `fgetc()` permite ler um único carácter por vez do ficheiro.

```
char c = fgetc(fp);
```

É útil quando se deseja processar o conteúdo carácter a carácter, como em contadores de letras, verificação de símbolos ou leitura personalizada.

d) `sscanf()`

A função `sscanf()` não lê diretamente do ficheiro, mas de uma *string* que já está em memória, normalmente obtida com fgets().

```
FILE *fp;
char linha[100];
fgets(linha, sizeof(linha), fp);
sscanf(linha, "%d %f", &inteiro, &decimal);
```

É especialmente útil para processar dados estruturados em linhas que já foram carregadas, permitindo maior controlo e evitando problemas de leitura causados por espaços em branco.

Cada uma dessas funções tem a sua aplicação ideal. Em geral, `fscanf()` é usada quando os dados são bem estruturados e formatados diretamente no ficheiro; `fgets()` é preferida para capturar textos mais livres (como linhas completas); e `fgetc()` é utilizada para leitura carácter a carácter, permitindo um controlo mais detalhado.

Um caso comum é combinar `fgets()` com `sscanf()`. Primeiro, usa-se `fgets()` para ler uma linha completa do ficheiro como string, e depois `sscanf()` para interpretar essa string, extraindo valores formatados (números, palavras etc.) de forma segura. Isso evita problemas que `fscanf()` pode causar quando há quebras de linha ou espaços inesperados, conforme pode ser visto no exemplo a seguir.

```c
FILE *fp;
char linha[100];
int inteiro;
float decimal;

fp = fopen("dados.txt", "r");
if (fp == NULL) {
    perror("Erro ao abrir ficheiro");
    return 1;
}

if (fgets(linha, sizeof(linha), fp) != NULL) {
    sscanf(linha, "%d %f", &inteiro, &decimal);
}

fclose(fp);
```

É importante sempre verificar se a leitura teve sucesso, especialmente ao chegar ao fim do ficheiro (`EOF`), usando funções como `feof()` ou verificando se `fgets()` retornou `NULL`.

13.2 Testes de Fim de Ficheiro

Quando se percorre um ficheiro para leitura, geralmente não se sabe quantas linhas ou registos contém. Por isso, é essencial saber quando parar. Em C, o fim de um ficheiro é indicado por um valor especial, o `EOF` (*End Of File*), usado por diversas funções de leitura para sinalizar que o conteúdo terminou. Além disso, a linguagem fornece a função `feof()`, que verifica se o apontador do ficheiro chegou ao final. No entanto, essa função só retorna verdadeiro após uma tentativa de leitura que falhou por ultrapassar o fim.

É importante destacar que, a cada leitura, o apontador `FILE` avança automaticamente para o dado seguinte, sem necessidade de qualquer incremento manual.

A seguir há alguns trechos de código como exemplos de como ler um ficheiro até o fim, utilizando diferentes funções.

13.2.1 Leitura com fscanf()

```
FILE *fp = fopen("dados.txt", "r");
//ler o ficheiro "dados.txt" no diretório atual
char nome[30];
int idade;

if (fp == NULL) {
    perror("Erro ao abrir o ficheiro");
    exit(1);
}
while (fscanf(fp, "%s %d", nome, &idade) == 2) {
//leitura de dois itens válidos (nome e idade)
    printf("Nome: %s, Idade: %d\n", nome, idade);
}
fclose(fp);
```

Neste exemplo, a leitura continua enquanto forem lidos dois itens válidos (`nome e idade`). Quando a estrutura do ficheiro muda ou o final é alcançado, o `while()` é interrompido.

13.2.2 Leitura com fgets()

```c
FILE *fp = fopen("D:\\linhas.txt", "r");
/* Para especificar caminhos no Windows, precisa usar
  dupla barra invertida \\ porque a barra invertida
  \ é um carácter de escape.  */
char linha[100];
if (fp == NULL) {
    perror("Erro ao abrir o ficheiro");
    exit(1);
}

while (fgets(linha, sizeof(linha), fp) != NULL) {
    printf("Linha: %s", linha);
}
fclose(fp);
```

Neste exemplo, `fgets()` retorna `NULL` quando chega ao fim do ficheiro. Essa forma é ideal para ficheiros onde cada linha representa uma informação completa.

13.2.3 Leitura com fgetc()

```c
FILE *fp = fopen("letra.txt", "r");
int c;
if (fp == NULL) {
    perror("Erro ao abrir o ficheiro");
    exit(1);
}
while ((c = fgetc(fp)) != EOF) {
    putchar(c);
}
fclose(fp);
```

Neste caso, `fgetc()` retorna um carácter de cada vez e `EOF` ao final. Essa abordagem é útil para processamentos mais finos, como contagem de letras ou análise de símbolos.

Observação sobre `feof()`

Evite usar `while (!feof(fp))` para controlar a leitura. Isso pode causar problemas, pois `feof()` só detecta o fim depois de uma tentativa malsucedida. O uso correto é testar diretamente o valor retornado pela função de leitura, como nos exemplos anteriores.

13.3 Fechamento do Ficheiro

Todo ficheiro aberto com `fopen()` deve ser fechado com `fclose(fp);` assim que o seu uso terminar. Isso garante que todos os dados em memória sejam efetivamente gravados no disco e que os recursos do sistema operativo sejam liberados, evitando vazamentos de memória ou travamentos.

```
fclose(fp);
```

O fechamento é especialmente importante em ficheiros abertos para escrita, pois o sistema normalmente mantém os dados em um buffer temporário na memória. Esses dados só são gravados fisicamente na unidade de armazenamento (como HD, SSD ou pendrive) quando o ficheiro é fechado corretamente com `fclose()`. Se isso não for feito, parte das informações pode ser perdida, mesmo que o programa tenha aparentemente terminado com sucesso.

Em situações mais específicas, como quando vários ficheiros foram abertos e o programa está a terminar, é possível usar `fcloseall();`, que tenta fechar todos os ficheiros abertos. No entanto, essa função não faz parte do padrão da linguagem C e não é suportada em todos os compiladores. Portanto, seu uso não é recomendado em programas portáveis.

A prática segura e recomendada é sempre fechar cada ficheiro individualmente com `fclose()`.

13.4 Ficheiros Binários

Ficheiros binários armazenam os dados no mesmo formato utilizado na memória do computador, *byte* a *byte*. Isso significa que não há conversão para caracteres legíveis, como ocorre em ficheiros de texto. Essa abordagem é útil para salvar diretamente estruturas, vetores ou qualquer bloco de dados com mais precisão e eficiência, evitando perdas de formatação ou limitações de representação.

Por serem gravados em formato bruto (binário), esses ficheiros não podem ser abertos e compreendidos por um editor de texto comum. Eles exigem que o programa saiba exatamente como os dados foram gravados, para que possa interpretá-los corretamente na leitura.

Assim como nos ficheiros texto, o trabalho com ficheiros binários começa com a abertura por meio da função `fopen()`. Os modos de abertura seguem os mesmos, com o acréscimo da letra `"b"` indicando acesso binário, conforme a Tabela 30.

Tabela 30 Modos de abertura de ficheiros binários

Modo	Descrição
"rb"	Leitura de ficheiro binário
"wb"	Escrita de ficheiro binário
"ab"	Adição no final de ficheiro binário
"rb+"	Leitura e escrita
"wb+"	Escrita e leitura (apaga antes)
"ab+"	Leitura e adição ao final

13.4.1 Entrada e saída dos dados no ficheiro binário

Por serem armazenados tal como estão na memória, os ficheiros binários exigem funções específicas para leitura e gravação. As funções `fwrite()` e `fread()` são utilizadas para isso, pois permitem copiar blocos inteiros de memória entre a RAM e o disco, sem realizar conversões para texto. Essas funções operam diretamente com apontadores genéricos e tamanhos de bloco, o que as torna ideais para gravar ou recuperar `structs` e `arrays`.

As funções `fprintf()` e `fscanf()`, assim como as demais funções anteriormente vistas para ficheiros texto, não devem ser usadas em ficheiros binários, pois realizam conversões para texto que alteram a forma original como os dados são representados na memória.

13.4.2 Leitura e Escrita em Ficheiros Binários

Para manipular dados em ficheiros binários, como já apresentado anteriormente, se utiliza principalmente as funções `fwrite()` e `fread()`.

A forma de uso de `fread()` é praticamente a mesma de `fwrite()`, com os mesmos parâmetros: endereço de memória, tamanho do tipo, quantidade de itens e apontador para o ficheiro, ou seja, a sintaxe é idêntica das duas funções, apenas mudando o sentido da operação.

Sintaxe para escrita no ficheiro:

```
fwrite(endereco, tamanho, quantidade, apontador);
```

Sintaxe para leitura no ficheiro:

```
fread(endereco, tamanho, quantidade, apontador);
```

onde

- `endereco`: apontador para a variável ou estrutura a ser gravada ou lida
- `tamanho`: número de bytes de cada elemento (geralmente obtido com `sizeof(tipo)`, que retorna a quantidade de memória (em bytes) ocupada pela variável.
- `quantidade`: número de elementos a ler ou escrever
- `apontador`: apontador para o ficheiro, obtido com `fopen()`

Essas funções devolvem a quantidade de elementos efetivamente escritos ou lidos em um ficheiro binário.

As funções `fwrite()` e `fread()` retornam quantos elementos foram gravados ou lidos com sucesso. Esse valor deve ser comparado com o número esperado. Se forem diferentes, pode ter ocorrido um erro ou o fim do ficheiro foi alcançado.

13.4.3 Exemplo – Gravar e Ler uma struct:

No exemplo a seguir, o programa declara duas variáveis do tipo `Pessoa`: `p1`, inicializada com o nome "`Maria Lopes`" e a idade `30`, e `p2`, que começa sem valores definidos. Em seguida, grava `p1` num ficheiro binário e depois lê esse ficheiro para preencher `p2`, exibindo no ecrã os dados lidos. Esse procedimento demonstra como gravar e recuperar estruturas em formato binário, preservando exatamente os valores originais, já que `p2` recebe os dados que estavam armazenados no ficheiro e passa a conter uma cópia idêntica de `p1`.

```c
typedef struct {
    char nome[50];
    int idade;
} Pessoa;

int main() {
    Pessoa p1 = {"Maria Lopes", 30}, p2;
    /* Declara duas variáveis do tipo Pessoa e inicia
       a variável p1  (grava em binário) */
    FILE *fp = fopen("dados.bin", "wb");
    if (fp != NULL) {
        fwrite(&p1, sizeof(Pessoa), 1, fp);
        fclose(fp);
    }
    // Ler do binário e exibir no ecrã
    fp = fopen("dados.bin", "rb");
    if (fp != NULL) {
        fread(&p2, sizeof(Pessoa), 1, fp);
        fclose(fp);
    }
    printf("Nome: %s\n", p2.nome);
    printf("Idade: %d\n", p2.idade);
    return 0;
}
```

Vale ressaltar que `p2` não era necessário, podendo usar apenas `p1`, pois não importa o nome da variável.

13.5 Gravação e leitura de uma struct

Para ilustrar como escrever e ler dados estruturados em ficheiros binários, se utiliza uma `struct` simples que representa uma pessoa. Esse exemplo mostra como gravar os dados diretamente da memória para o disco e recuperá-los posteriormente, mantendo a integridade da informação.

```c
typedef struct {
        char nome[50];
        int idade;
} Pessoa;
```

Gravar uma `struct` em ficheiro binário

```c
Pessoa p = {"Maria", 30};

FILE *fp = fopen("dados.bin", "wb");
if (fp != NULL) {
    fwrite(&p, sizeof(Pessoa), 1, fp);
     // grava 1 bloco do tamanho da struct
    fclose(fp);
}
```

13.5.1 Ler uma struct de um ficheiro binário

```c
Pessoa p2;

FILE *fp = fopen("dados.bin", "rb");
if (fp != NULL) {
    fread(&p2, sizeof(Pessoa), 1, fp);
// Ler 1 bloco do tamanho da struct
    fclose(fp);
}
```

A função `fwrite()` recebe como argumentos:

1. o endereço da variável a ser gravada
2. o tamanho de cada bloco (geralmente `sizeof(struct)`)
3. o número de blocos
4. o apontador para o ficheiro

Já `fread()` possui a mesma lógica, mas no sentido contrário, ou seja, lê os dados do ficheiro e os coloca diretamente na variável fornecida.
Essas funções garantem que os dados estruturados sejam mantidos com exatidão, sem perdas por formatação, facilitando o armazenamento e a recuperação de informações complexas.

Observação: Se há no ficheiro um vetor com várias estruturas (por exemplo, 100 elementos), é possível ler todo o vetor de uma só vez alterando o terceiro parâmetro da função `fread()`, como no exemplo a seguir.

```
Pessoa lista[100];
fread(lista, sizeof(Pessoa), 100, fp);
```

Essa abordagem é eficiente e facilita a leitura em bloco de grandes volumes de dados.
A mesma ideia serve para gravar em ficheiro binário com `fwrite()`.
É muito importante saber que esse tipo de leitura ou gravação só é válido para ficheiros binários. Não é possível fazer o mesmo com ficheiros de texto nem imprimir diretamente esse conteúdo binário no ecrã com `printf()`.

13.5.2 Leitura até o fim de um ficheiro binário

Ao trabalhar com ficheiros binários, muitas vezes não se sabe quantos dados estão armazenados. Para percorrer todo o conteúdo, utiliza-se um laço que continua a leitura enquanto a função `fread()` conseguir obter os dados

corretamente. A abordagem mais segura é testar o valor retornado por `fread()`, que indica o número de itens lidos com sucesso.

A seguir se apresenta um exemplo que lê e exibe, um a um, todos os registos de um ficheiro binário contendo estruturas do tipo `Pessoa`:

```
typedef struct {
    char nome[50];
    int idade;
} Pessoa;
int main() {
    FILE *fp = fopen("dados.bin", "rb");
    Pessoa p;
    if (fp == NULL) {
        perror("Erro ao abrir o ficheiro");
        return 1;
    }

    while (fread(&p, sizeof(Pessoa), 1, fp) == 1) {
        printf("%s: %d anos\n", p.nome, p.idade);
    }

    fclose(fp);
    return 0;
}
```

Neste caso, se está a ler uma estrutura por vez, por isso se usa `1` como terceiro argumento de `fread()`. Isso indica que se quer ler 1 item, cada um com tamanho igual a `sizeof(Pessoa)`.

Mas é importante perceber que se usa a comparação com o valor `1` no `fread()` e `2` com `fscanf()`, porque as funções de leitura retornam o número de itens lidos com sucesso. Isso varia conforme o tipo de função e a quantidade de dados que se espera ler. Em `fread()`, o controlo é feito

com blocos de memória, por isso é comum ler um bloco por vez e testar se ele foi realmente lido. Em `fscanf()`, por outro lado, testa-se quantas variáveis formatadas foram capturadas corretamente, geralmente correspondendo ao número de especificadores na expressão de controlo (`%s %d`, por exemplo).

Essa comparação é muito importante para garantir que percebeste e fixaste como é o controlo da leitura correcta nos dois tipos de ficheiros, garantindo segurança e evitando leituras incompletas.

13.6 Vantagens e Desvantagens dos ficheiros binários

Os ficheiros binários oferecem uma forma eficiente de armazenar dados exatamente como estão representados na memória do computador, sem conversões para texto. Isso traz diversas vantagens: a leitura e escrita são mais rápidas, especialmente quando se lida com grandes volumes de informação; o espaço em disco é mais bem aproveitado, pois os dados ocupam menos *bytes* do que em formato textual; e estruturas completas, como *structs*, podem ser salvas diretamente, preservando todos os seus campos com precisão.

No entanto, o uso de ficheiros binários também apresenta algumas desvantagens importantes. Como os dados não são legíveis por humanos, não é possível abri-los com um editor de texto comum para inspecionar ou alterar seu conteúdo, sendo necessário um programa que saiba interpretar exatamente a estrutura usada.

Além disso, qualquer alteração na definição da `struct` original (como trocar a ordem dos campos ou adicionar um novo campo) pode tornar o ficheiro binário inválido ou incompatível com versões anteriores. Por fim, há a questão da portabilidade: ficheiros binários podem ter comportamentos diferentes em máquinas com arquiteturas distintas, como sistemas de 32 ou 64 bits ou com diferentes formas de organização dos bytes (*endianness*), dificultando o uso em plataformas variadas.

13.7 Exemplos de leitura e escrita em ficheiros

A seguir, apresentam-se exemplos simples e comentados de como gravar e ler dados em ficheiros de texto e binários, além de uma função que implementa persistência de dados usando ficheiros binários.

13.7.1 Gravar dados num ficheiro de texto

Neste exemplo, cria (ou sobrescreve) o ficheiro `pessoas.txt` e grava nele dois registos em formato legível.

```
FILE *fp = fopen("pessoas.txt", "w");
if (fp != NULL) {
    fprintf(fp, "Ana 25\nCarlos 30\n");
    fclose(fp);
} else {
    perror("Erro ao abrir ficheiro");
}
```

13.7.2 Ler um vetor de estruturas de ficheiro binário

Aqui, se lê 10 registos do tipo `Pessoa` gravados previamente em `dados.bin`. Atenção para o fato de que a `struct Pessoa` deve estar definida antes desta leitura

```
Pessoa pessoas[10];
FILE *fp = fopen("dados.bin", "rb");
if (fp != NULL) {
    fread(pessoas, sizeof(Pessoa), 10, fp);
    fclose(fp);
}
```

13.7.3 Função com gravação de dados (persistência)

Esta função recebe um vetor de `Pessoa` e o grava no ficheiro binário `dados.dat`, permitindo que os dados fiquem disponíveis mesmo após fechar o programa.

```
void salvarDados(Pessoa lista[], int tamanho) {
    FILE *fp = fopen("dados.dat", "wb");
    if (fp != NULL) {
        fwrite(lista, sizeof(Pessoa), tamanho, fp);
        fclose(fp);
    } else {
        perror("Erro ao abrir ficheiro");
    }
}
```

13.7.4 Leitura de ficheiro texto com fgets() + sscanf()

Este exemplo mostra como ler linhas completas de um ficheiro de texto e extrair os dados formatados de forma segura, evitando problemas com nomes compostos.

```
if (fp != NULL) {
    while (fgets(linha, sizeof(linha), fp) != NULL) {
        // Remove \n, se existir
        linha[strcspn(linha, "\n")] = '\0';

        // Extrai dados (nome sem espaços neste exemplo)
        if (sscanf(linha, "%s %d", p.nome, &p.idade) == 2) {
            printf("Nome: %s | Idade: %d\n", p.nome, p.idade);
        } else {
            printf("Linha inválida: %s\n", linha);
        }
    }
    fclose(fp);
} else {
    perror("Erro ao abrir ficheiro");
}
```

13.8 Manipulação Avançada de Ficheiros

Além das funções básicas de leitura e escrita, a linguagem C oferece mecanismos mais avançados para controlar o acesso aos ficheiros.

13.8.1 fseek()

A função `fseek()`, que permite mover o cursor de leitura/escrita para qualquer posição do ficheiro, sem precisar percorrer todo o conteúdo desde o início, cuja sintaxe e:

```
fseek(fp, deslocamento, origem);
```

onde:

- `fp`: apontador para o ficheiro (já aberto)
- deslocamento: número de bytes a mover
- origem: ponto de referência (pode ser o início, a posição atual ou o fim do ficheiro). As opções de origem são:
    - `SEEK_SET`: início do ficheiro
    - `SEEK_CUR`: posição atual
    - `SEEK_END`: fim do ficheiro

Um exemplo do uso do `fseek()` está apresentado a seguir:

```
fseek(fp, 0, SEEK_SET);
// Posiciona no início do ficheiro
fseek(fp, sizeof(Pessoa)*3, SEEK_SET);
// Vai direto ao 4º registo (índice 3)
```

A função `fseek()` é especialmente útil em ficheiros binários, onde cada registo tem tamanho fixo e é possível calcular diretamente sua posição.

13.8.2 Exportação de Dados para Impressão ou Planilha

Em aplicações modernas em C, é comum exportar os dados para ficheiros auxiliares que depois podem ser abertos por outros programas (como planilhas ou para um leitor de PDF). A impressão direta (como no antigo `PRN`) foi substituída por formatos portáteis e acessíveis.

a) Exportar para ficheiro .txt (texto simples)

Permite gerar relatórios legíveis e prontos para serem impressos por qualquer editor de texto:

```
FILE *fp = fopen("relatorio.txt", "w");
if (fp != NULL) {
    fprintf(fp, "Relatório de Vendas\n");
    fprintf(fp, "-------------------\n");
    fprintf(fp, "Produto A: 120 unidades\n");
    fprintf(fp, "Produto B: 85 unidades\n");
    fclose(fp);
}
```

O utilizador pode abrir esse ficheiro no Bloco de Notas, editores de texto ou similar e imprimir normalmente.

b) Exportar para `.csv` (valores separados por vírgula)

O formato `.csv` é amplamente reconhecido por programas como Microsoft Excel, Google Sheets ou LibreOffice Calc.

Ele organiza os dados em linhas e colunas, facilitando análise e visualização:

```
FILE *fp = fopen("dados.csv", "w");
if (fp != NULL) {
    fprintf(fp, "Produto,Quantidade,Preço\n");
    fprintf(fp, "Camisola,50,19.99\n");
    fprintf(fp, "Calça,30,39.90\n");
    fclose(fp);
}
```

Esse ficheiro pode ser aberto num editor de planilhas com separação automática das colunas.

c) Exportar para .xls (binário)

A linguagem C não possui suporte direto para o formato binário `.xls`, do MS Excel, mas há bibliotecas externas que permitem criar ficheiros `.xls` ou `.xlsx` diretamente.

A alternativa prática é gerar um `.csv` bem formatado e abrir no Excel, que permite salvar como `.xls/.xlsx`.

d) Exportar para .pdf

O C padrão também não gera `.pdf` diretamente, mas também há bibliotecas externas que permitem isso.

Como alternativa, pode-se gerar um `.txt`, `.csv` ou `.html` e usar ferramentas externas para converter em PDF.

Em resumo, ao escolher o formato de ficheiro para guardar ou trocar informações, é importante considerar as características e limitações de cada um, como:

- .txt: simples e legível, ideal para relatórios.
- .csv: estruturado, ideal para planilhas e análise de dados.
- .pdf: profissional e portátil, exige bibliotecas externas.
- .xls/.xlsx: requer bibliotecas específicas, mas pode ser substituído por .csv.

Evite usar `fopen("PRN", "w")` ou `stdprn`, pois são métodos ultrapassados, limitados a sistemas muito antigos.

13.9 Exercícios Propostos

1. Lista de Séries Assistidas*:* Com tantas plataformas de *streaming*, é interessante registar o que já foi visto. Para isso, crie um ficheiro de texto contendo uma lista de séries, a ser informada pelo utilizador e, em seguida, mostrar o conteúdo formatado no ecrã. O programa deve:

- Ler do teclado o nome de **5 séries** (usar `fgets()`
- Gravar cada série, uma por linha, no ficheiro `series.txt.`
- Ler o ficheiro e exibir a lista formatada no ecrã.

Dicas:

- Remova o `\n` do final das strings.
- Utilize `fprintf()` para escrever no ficheiro e `fgets()` para ler.

Protótipo sugerido:

```
void gravarSeries(const char *ficheiro);
void lerSeries(const char *ficheiro);
```

2. Ranking de Jogos Favoritos: Para que os *gamers* possam comparar pontuações, faça um programa para criar um ranking simples que pode ser usado para organizar torneios. Para tal, registe jogos com nome e pontuação, guarde num ficheiro binário e exiba o ranking ordenado. Seu programa deve:

- Definir `struct Jogo` com campos `nome` (até 50 caracteres) e `pontuacao` (`int`)
- Ler 5 jogos e respetivas pontuações (de 0 até 10)
- Gravar no ficheiro binário `jogos.bin` usando `fwrite()`
- Ler o ficheiro, ordenar por pontuação (maior para menor) e exibir

Protótipos sugerido:

```
void gravarJogos(const char *ficheiro, struct Jogo jogos[], int n);
void lerJogos(const char *ficheiro, struct Jogo jogos[], int n);
int compararJogos(const void *a, const void *b);
```

3. Conversor de Contactos: Muitos contactos estão guardados em ficheiros de texto exportados de telemóveis ou emails. Faça um programa para criar um conversor para um formato binário mais rápido e seguro, ou seja, ler contactos de um ficheiro texto e convertê-los para binário. Tarefas:

- Criar `struct Contacto` com `nome[50] e telefone[15]`
- Ler ficheiro texto `contactos.txt`, onde cada linha tem formato:

```
Nome Completo;912345678
```

- Usar `fgets()` para ler linhas e `sscanf()` para separar `nome` e `telefone`
- Guardar num ficheiro binário `contactos.dat`
- Ler o ficheiro binário e exibir os contactos.

Dicas:

- `sscanf(linha, "%49[^;];%14s", nome, telefone);`
- Validar se o ficheiro de texto foi aberto antes de ler.

Protótipo sugerido:

```
int lerContactosTexto(const char *ficheiroTexto, struct Contato contactos[], int max);
void gravarContactosBinario(const char *ficheiroBin, struct Contato contactos[], int n);
```

```
void lerContactosBinario(const char *ficheiroBin,
struct Contato contactos[], int n);
```

4. Relatório de Vendas com Backup: Um sistema de vendas armazena dados em binário, mas o gestor precisa de relatórios em texto para análise e um backup seguro dos dados.

Faça um programa para ler vendas de um ficheiro binário, criar um relatório de texto e fazer um backup binário. Tarefas:

- Criar `struct Venda` com `produto[50]`, `quantidade` (`int`) e `preco (float)`.
- Ler todos os registos do ficheiro binário `vendas.dat`.
- Calcular o `total` (`quantidade × preco`) de cada produto.
- Criar o ficheiro `relatorio.txt` com colunas alinhadas: `produto, quantidade, preço e total`.
- Criar também `backup_vendas.dat` com os registos originais.

Dicas:

- Formatar com `fprintf(fp, "%-20s %5d %10.2f %10.2f\n", ...)`.
- Usar `fread()` e `fwrite()` para binários.

Protótipos sugerido:

```
int lerVendasBinario(const char *ficheiro, struct Venda
vendas[], int max);
void gerarRelatorioTexto(const char *ficheiroTexto,
struct Venda vendas[], int n);
void criarBackupBinario(const char *ficheiroBackup,
struct Venda vendas[], int n);
```

14 CRIAR BIBLIOTECA EM C

À medida que os programas crescem, torna-se comum reutilizar funções e trechos de código em várias partes do projeto. Para organizar melhor o código e permitir essa reutilização de forma simples, a linguagem C permite criar bibliotecas personalizadas.

Neste capítulo, está a ser apresentado como construir uma biblioteca própria em C e utilizá-la noutros programas.

14.1 O que é uma biblioteca?

Uma biblioteca em C é um conjunto de funções agrupadas em ficheiros separados. Essas funções podem ser reutilizadas em vários programas, bastando incluir a biblioteca desejada no código principal.

Criar uma biblioteca em C é uma forma prática e inteligente de organizar funções que já foram testadas e funcionam bem, para que possam ser reutilizadas em diferentes programas. Ao invés de copiar e colar o mesmo código várias vezes, coloca-se essas funções num único sítio, na biblioteca, e depois apenas as chamando onde forem necessárias.

Essa abordagem traz várias vantagens:

- Evita repetição: funções comuns como cálculos, validações ou leitura de dados podem ser escritas uma única vez e usadas em qualquer programa.
- Facilita a manutenção: se for necessário corrigir um erro ou melhorar uma função, basta mudar na biblioteca — todos os programas que usam essa função serão automaticamente beneficiados.

- Melhora a organização: separa as funções auxiliares do código principal, tornando o programa mais limpo e fácil de entender.
- Permite reutilização: ao longo do tempo, pode criar sua própria coleção de funções úteis, que servem de base para projetos futuros.

Em resumo, uma biblioteca é um conjunto de funções e recursos já prontos, criados para serem reutilizados em diferentes programas. Ao utilizá-la, evita-se reescrever o mesmo código várias vezes, tornando o desenvolvimento mais rápido, organizado e eficiente.

14.2 Estrutura de uma biblioteca

Uma biblioteca simples em C é composta por dois ficheiros:

- Um ficheiro de cabeçalho (`.h`), onde se declara as funções
- Um ficheiro de implementação (`.c`), onde se escreve o código dessas funções

Esses ficheiros são criados separadamente e utilizados em conjunto com o programa principal (`main.c`).

14.3 Exemplo completo de biblioteca

Neste exemplo, se está a criar uma biblioteca chamada `util.h` e `util.c`, com funções para ler e mostrar números inteiros.

Os ficheiros `.h` (*headers*) servem para declarar funções, estruturas, constantes e outros elementos que serão usados em vários ficheiros do programa. Porém, ao incluir o mesmo ficheiro `.h` em mais de um sítio, pode

acontecer de ele ser incluído duas vezes, causando erros de compilação por redefinição de elementos (como funções ou estruturas já declaradas).
Para evitar esse problema, se usa um conjunto especial de diretivas de pré-processador: `#ifndef, #define e #endif`. Essas diretivas formam o que se chama de guarda de inclusão (*include guard*).

Em C, existem duas formas de incluir ficheiros com a diretiva `#include`:

```
#include <ficheiro.h>
#include "ficheiro.h"
```

A diferença está no caminho de procura:

- `<ficheiro.h>`: o compilador procura apenas nas pastas padrão do sistema, onde estão localizadas as bibliotecas da linguagem C, como `stdio.h, stdlib.h, math.h` etc.
- `"ficheiro.h"`: o compilador procura primeiro na pasta do projeto, onde está o código-fonte. Se não encontrar, só então procura nas pastas padrão. É a forma correta de incluir ficheiros criados por si, como `util.h`, a seguir, que está na mesma pasta do projeto.

Em resumo:

- `#include "ficheiro.h"`: para ficheiros do seu projeto.
- `#include <ficheiro.h>`: para bibliotecas padrão do compilador.

Essa convenção ajuda o compilador a encontrar os ficheiros corretamente além de manter o código mais organizado.

a) Ficheiro de cabeçalho: util.h

```
#ifndef UTIL_H
#define UTIL_H
void ler_numeros(int v[], int n);
void mostrar_numeros(int v[], int n);

#endif
```

- Neste ficheiro, apenas se declara as funções
- As diretivas `#ifndef`, `#define` e `#endif` que evitam que o ficheiro seja incluído duas vezes por engano.

b) Ficheiro de implementação: util.c

```
#include <stdio.h>
#include "util.h"

void ler_numeros(int v[], int n) {
    for (int i = 0; i < n; i++) {
        printf("Digite o número %d: ", i + 1);
        scanf("%d", &v[i]);
    }
}

void mostrar_numeros(int v[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", v[i]);
    }
    printf("\n");
}
```

- Neste ficheiro, se implementa as funções declaradas em `util.h`.
- Usa-se `#include "util.h"` para ligar às funções ao cabeçalho.

c) Programa principal: main.c

```c
#include <stdio.h>
#include "util.h"

int main() {
    int numeros[5];

    ler_numeros(numeros, 5);
    mostrar_numeros(numeros, 5);

    return 0;
}
```

O programa principal usa a biblioteca apenas incluindo o cabeçalho com #include "util.h".

14.4 Compilação dos ficheiros

Já se sabe que ao desenvolver programas de maior dimensão ou que necessitem de organização clara, é recomendável dividir o código em múltiplos ficheiros. Isso facilita a manutenção, reutilização e leitura do projeto.

Neste exemplo, foi criado um projeto com três ficheiros:

- `main.c` – Contém a função `main()` e a lógica principal do programa.
- `util.c` – Implementa funções auxiliares (módulo de utilidades).
- `util.h` – Ficheiro de cabeçalho que declara as funções e constantes usadas por outros ficheiros.

Para compilar, basta executar normalmente o comando de compilação incluindo todos os ficheiros fonte. O compilador irá processar cada um individualmente e, em seguida, o *linker* irá juntar todos os módulos num único executável.

14.5 Organização das bibliotecas

Para reutilizar a biblioteca em vários projetos:

- Guarde os seus `.h` e `.c` numa pasta separada (exemplo: `minhas_bibliotecas`)
- Quando criar um novo projeto, adicione essa pasta aos diretórios de inclusão do compilador, se necessário.

Assim, poderá usar suas funções em qualquer programa sem precisar reescrevê-las.

Criar bibliotecas próprias é uma prática comum e recomendada em projetos reais, pois traz diversos benefícios importantes. Ao reunir funções reutilizáveis num único sítio, reduz-se a possibilidade de erros causados por duplicação ou inconsistência de código. Além disso, a clareza do programa melhora significativamente, uma vez que o código principal fica mais limpo e focado na lógica geral, enquanto os detalhes de implementação são organizados à parte. Outro ponto positivo é que o desenvolvimento se torna mais rápido e estruturado, já que as bibliotecas podem ser reaproveitadas em diferentes projetos ou partes do programa. Por fim, ao separar o código em

módulos distintos, é possível dividir o trabalho entre várias pessoas da equipa, facilitando a colaboração e a manutenção do projeto como um todo.

14.6 Bibliotecas Úteis em C

A linguagem C possui um conjunto robusto de bibliotecas padrão que fornecem funções prontas para resolver tarefas comuns. Ao incluir essas bibliotecas com `#include`, se pode aceder a funcionalidades matemáticas, manipulação de cadeias de caracteres, alocação dinâmica, tratamento de datas e muito mais, sem ter de reinventar a roda.
Este capítulo apresenta algumas das bibliotecas mais utilizadas em C, com exemplos práticos de aplicação para que compreenda melhor como e quando usá-las.

14.6.1 math.h – Funções Matemáticas

A biblioteca `math.h` oferece funções matemáticas avançadas como raízes quadradas, potências, funções trigonométricas, logaritmos, entre outras.
Funções comuns:

- `sqrt(x)` – retorna a raiz quadrada de x
- `pow(x, y)` – calcula x elevado à y
- `sin(x), cos(x)` – funções trigonométricas (entrada em radianos)

Exemplos de uso:

```c
num = 100;
double raiz = sqrt(num);
double potencia = pow(2, 3);
// 2 elevado ao cubo
```

14.6.2 string.h – Manipulação de Strings

Esta biblioteca fornece funções para comparar, copiar, medir e alterar strings (cadeias de caracteres terminadas por \0).

Algumas funções:

- `strlen(s)` – retorna o comprimento da string s
- `strcpy(dest, src)` – copia o conteúdo de `src` para `dest`
- `strcmp(s1, s2)` – compara duas strings

Exemplos de uso:

```c
char senha[10];
strcpy(senha, "123");
if (strcmp(senha, "123") == 0) {
    printf("Acesso permitido.\n");
}
```

14.6.3 ctype.h – Classificação de Caracteres

Com `ctype.h`, é possível testar e converter caracteres individuais: verificar se são letras, dígitos, maiúsculas, etc.

Funções comuns:

- `isalpha(c)` – verifica se `c` é uma letra

- `isdigit(c)` – verifica se `c` é um dígito
- `toupper(c)` – converte para maiúscula

Exemplo:

```c
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'a';

    if (isalpha(c)) {
        printf("%c é uma letra\n", c);
        printf("Maiúscula: %c\n", toupper(c));
    }
    return 0;
}
```

14.6.4 stdlib.h – Utilitários Diversos

A biblioteca `stdlib.h` oferece funções para alocação de memória, geração de números aleatórios, conversão de strings para inteiros, entre outras.

Funções comuns:

- `malloc(tamanho)` – aloca memória
- `rand()` – gera um número aleatório
- `atoi(s)` – converte uma string para `int`
- `free(ptr)` – liberta um bloco de memória previamente alocado com `malloc()`, `calloc()` ou `realloc()`
- `system(comando)` – executa um comando do sistema operativo a partir do programa em C.

Exemplo:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *v = malloc(5 * sizeof(int));
    if (v == NULL) return 1;

    for (int i = 0; i < 5; i++) {
        v[i] = rand() % 100;
        printf("%d ", v[i]);
    }

    free(v); // libera a memória
    return 0;
}
```

14.6.5 time.h – Tempo e Data

Permite trabalhar com tempo e calendário. É útil para gerar os valores aleatórios, medir tempo de execução ou exibir a data atual.

Funções comuns:

- `time(NULL)` – obtém o tempo atual (em segundos desde 1970)
- `localtime(&t)` – converte para data legível
- `strftime(buffer, tamanho, formato, &tm)` – formata a data/hora numa *string* de acordo com um formato especificado
- `clock()` – obtém o tempo de CPU usado pelo programa, útil para medir tempo de execução.

Exemplo:

```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    srand(time(NULL)); // semente para rand()
    int aleatorio = rand() % 100;
    printf("Número aleatório: %d\n", aleatorio);

    time_t agora = time(NULL);
    struct tm *info = localtime(&agora);
    printf("Data: %02d/%02d/%04d\n",
           info->tm_mday, info->tm_mon + 1,
           info->tm_year + 1900);
    return 0;
}
```

14.7 Bibliotecas Padrão

Existem muitas bibliotecas padrão em C, com funções específicas para diferentes finalidades.

Essas bibliotecas facilitam o desenvolvimento, reduzem a necessidade de reinventar soluções e tornam o código mais confiável, legível e reutilizável.

Todas fazem parte do padrão da linguagem C e estão disponíveis em qualquer compilador compatível.

A lista completa e a descrição oficial de cada biblioteca podem ser encontradas na documentação da norma C (ISO/IEC 9899) e também em manuais e referências online.

A Tabela 31 com as principais bibliotecas padrão do C e uma breve descrição de cada uma.

Tabela 31 Bibliotecas Padrão ANSI C

Biblioteca	Descrição resumida
`assert.h`	Macros para depuração e verificação de condições (*assert*).
`ctype.h`	Funções para teste e conversão de caracteres (isdigit, toupper, etc.).
`errno.h`	Define macros para relatórios e tratamento de erros (errno).
`float.h`	Limites e propriedades de números de ponto flutuante.
`limits.h`	Limites e propriedades de tipos inteiros (INT_MAX, CHAR_MIN, etc.).
`locale.h`	Configuração de localização (*locale*) para formatação de dados.
`math.h`	Funções matemáticas comuns (sqrt, sin, pow, etc.).
`setjmp.h`	Macros para salto não-local na execução do programa.
`signal.h`	Manipulação de sinais (*signals*) como interrupções e abortos.
`stdarg.h`	Manipulação de listas de argumentos variáveis em funções.
`stdbool.h`	Definição do tipo booleano (bool, true, false).
`stddef.h`	Definições comuns como NULL, size_t e offsetof.
`stdio.h`	Entrada e saída padrão (ficheiros, teclado, ecrã).
`stdlib.h`	Funções utilitárias (memória dinâmica, conversões, aleatórios, sistema).
`string.h`	Manipulação de *strings* e blocos de memória (strcpy, memcpy, etc.).
`time.h`	Manipulação de tempo e data.
`complex.h`	Operações com números complexos (desde C99).
`tgmath.h`	Macros para funções matemáticas genéricas (desde C99).
`inttypes.h`	Tipos inteiros de largura definida e macros associadas (desde C99).
`stdint.h`	Tipos inteiros padronizados (int32_t, uint8_t, etc.).
`wchar.h`	Manipulação de caracteres largos (*wide characters*).
`wctype.h`	Teste e conversão de caracteres largos.
`fenv.h`	Controle do ambiente de ponto flutuante (desde C99).

15 PROPOSTAS DE RESOLUCAO

15.1 Capítulo 4

4.1.

```
#include <stdio.h>

int main() {
    char nome[50];
    int idade;
    float altura, peso;

    printf("Nome (sem espacos): ");
    scanf("%49s", nome);
    printf("Idade (anos): ");
    scanf("%d", &idade);
    printf("Altura (m): ");
    scanf("%f", &altura);
    printf("Peso (kg): ");
    scanf("%f", &peso);

    printf("\nNome: %s\n", nome);
    printf("Idade: %d anos\n", idade);
    printf("Altura: %.2f m\n", altura);
    printf("Peso: %.1f kg\n", peso);
    return 0;
}
```

4.2.

```
#include <stdio.h>

int main() {
    int a, b;
    printf("Introduza dois inteiros: ");
```

```
    scanf("%d %d", &a, &b);

    printf("Soma: %d\n", a + b);
    printf("Diferenca: %d\n", a - b);
    printf("Produto: %d\n", a * b);
    if (b != 0)
       printf("Divisao: %.2f\n", (double)a / b);
    else
       printf("Divisao: indefinida (por zero)\n");
    return 0;
}
```

4.3.

```
#include <stdio.h>

int main() {
    char ch;
    float x;

    printf("Introduza um caractere: ");
    scanf(" %c", &ch);
    // espaço antes de %c ignora '\n' pendente
    printf("Introduza um numero decimal: ");
    if (scanf("%f", &x) != 1) {
        printf("Erro (texto nao numerico?).\n");
    } else {
        printf("Leu: ch='%c' e x=%f\n", ch, x);
    }
    return 0;
}
```

4.4.

```
#include <stdio.h>

int main() {
    const char dados[] = "Maria 30 1.60 55.2";
    char nome[50];
    int idade;
    float altura, peso;
```

```
    if (sscanf(dados, "%49s %d %f %f", nome, &idade,
               &altura, &peso) == 4) {
        printf("Nome: %s\nIdade: %d\nAltura: %.2f\n
               Peso: %.1f\n",
               nome, idade, altura, peso);
    } else {
        printf("Falha no sscanf.\n");
    }
    return 0;
}
```

4.5.

```
#include <stdio.h>

int main() {
    const char s[] = "Chave=9876";
    int numero;
    if (sscanf(s, "Chave=%d", &numero) == 1)
        printf("Numero: %d\n", numero);
    else
        printf("Falha no sscanf.\n");
    return 0;
}
```

4.6.

```
#include <stdio.h>

int main() {
    const char s[] = "Lisboa, Freguesia Alvalade";
    char cidade[50], freguesia[50];

    if (sscanf(s, " %49[^,], %*s %49s", cidade,
        freguesia) == 2) {
        printf("Cidade: %s\n", cidade);
        printf("Freguesia: %s\n", freguesia);
    } else {
        printf("Falha no sscanf.\n");
    }
    return 0;
```

```
}
```

4.7.

```
#include <stdio.h>

int main() {
    const char email[] = "ana.rosa@gmail.com";
    char dominio[100];

    if (sscanf(email, "%*[^@]@%99s", dominio) == 1)
        printf("Utilizador: %s\n", dominio);
        // imprime "gmail.com"
    else
        printf("Falha no sscanf.\n");
    return 0;
}
```

4.8.

```
#include <stdio.h>

int main() {
    char frase[200];
    printf("Escreva uma frase: ");
    if (fgets(frase, sizeof(frase), stdin) != NULL)
        printf("Voce escreveu: %s", frase);
    else
        printf("Falha ao ler.\n");
    return 0;
}
```

4.9.

```
#include <stdio.h>

int main() {
    const char s[] = "Aluno: Carlos Andrade Silva";
    char ultimo[50];
```

```
    if (sscanf(s, "Aluno: %*s %*s %49s", ultimo) == 1)
        printf("Ultimo nome: %s\n", ultimo);
               // Silva
    else
        printf("Falha no sscanf.\n");
    return 0;
}
```

15.2 Capítulo 5

5.1.

```
#include <stdio.h>

int main() {
    float pesoTotal;
    printf("Peso total no elevador (kg): ");
    scanf("%f", &pesoTotal);

    if (pesoTotal > 450.0) {
        printf("Peso excedido!\n");
    } else {
        printf("Dentro do limite.\n");
    }
    return 0;
}
```

5.2.

```
#include <stdio.h>

int main() {
    float t;
    printf("Tempo do corredor (s): ");
    scanf("%f", &t);

    if (t <= 0.0f) {
        printf("Tempo invalido.\n");
    } else if (t < 10.0) {
        printf("Atleta profissional!\n");
    } else if (t < 12.0) {
        printf("Muito bom!\n");
    } else if (t <= 15.0) {
        printf("Regular\n");
    } else {
        printf("Treines mais\n");
```

```
    }
    return 0;
}
```

5.3.

```
#include <stdio.h>

int main() {
    int a, b, opcao;
    printf("Introduza dois inteiros: ");
    scanf("%d %d", &a, &b);
    printf("Escolha a operacao (1-Soma, 2-Subtracao,
           3-Multiplicacao, 4-Divisao): ");
    scanf("%d", &opcao);

    switch (opcao) {
        case 1:
            printf("Resultado: %d\n", a + b);
            break;
        case 2:
            printf("Resultado: %d\n", a - b);
            break;
        case 3:
            printf("Resultado: %d\n", a * b);
            break;
        case 4:
            if (b == 0) {
                printf("Impossivel dividir por
zero.\n");
            } else {
                printf("Resultado: %.2f\n",
                       (double)a/ b);
            }
            break;
        default:
            printf("Opcao invalida.\n");
    }
    return 0;
}
```

5.4.

```
#include <stdio.h>

int main() {
    float valor;
    printf("Valor da compra (€): ");
    scanf("%f", &valor);

    if (valor >= 50.0)
        printf("Parque gratis.\n");
    else
        printf("Valor: 2€.\n");

    return 0;
}
```

5.5.

```
#include <stdio.h>

int main() {
    float compra, desconto = 0.0, total;
    int percent = 0;

    printf("Valor da compra (€): ");
    scanf("%f", &compra);

    if (compra <= 100.0) {
        percent = 0;
    } else if (compra <= 200.0) {
        percent = 5;
    } else {
        percent = 10;
    }

    desconto = compra * percent / 100.0;
    total = compra - desconto;

    printf("Desconto: %d%%\n", percent);
    printf("Total a pagar: %.2f€\n", total);
```

```
    return 0;
}
```

5.6.

```
#include <stdio.h>

int main() {
    float t;
    printf("Tempo do corredor (s): ");
    scanf("%f", &t);

    if (t <= 0.0) {
        printf("Tempo invalido.\n");
    } else if (t < 10.0) {
        printf("Atleta profissional!\n");
    } else if (t < 12.0) {
        printf("Muito bom!\n");
    } else if (t <= 15.0) {
        printf("Regular\n");
    } else {
        printf("Treines mais\n");
    }
    return 0;
}
```

5.7.

```
#include <stdio.h>

int main() {
    float v;
    printf("Velocidade do carro (km/h): ");
    scanf("%f", &v);

    if (v > 120.0)
        printf("Multa por excesso de velocidade.\n");
    else
        printf("Velocidade dentro do limite.\n");
```

```
    return 0;
}
```

5.8.

```
#include <stdio.h>

int main() {
    int heroi, inimigo;

    printf("Forca do heroi (0 a 100): ");
    scanf("%d", &heroi);
    printf("Forca do inimigo (0 a 100): ");
    scanf("%d", &inimigo);

    if (heroi > inimigo)
        printf("Vitoria! O inimigo foi derrotado.\n");
    else if (heroi == inimigo)
        printf("Empate! Ambos recuam.\n");
    else
        printf("Derrota! Precisas treinar mais.\n");

    return 0;
}
```

15.3 Capítulo 6

6.1.

```
#include <stdio.h>

int main() {
    int i;

    printf("Usando for():\n");
    for (i = 1; i <= 100; i++) {
        printf("%d ", i);
    }

    printf("\n\nUsando while():\n");
    i = 1;
    while (i <= 100) {
        printf("%d ", i);
        i++;
    }

    printf("\n");
    return 0;
}
```

6.2.

```
#include <stdio.h>

int main() {
    int i, valor;
    int soma = 0;
    float media;

    for (i = 1; i <= 10; i++) {
        printf("Introduza o %do numero inteiro: ", i);
```

```
        scanf("%d", &valor);
        soma += valor;
    }

    media = soma / 10.0;
    printf("Media = %.2f\n", media);

    return 0;
}
```

6.3.

```
#include <stdio.h>

int main() {
    int n, i, primo = 1;

    printf("Introduza um numero inteiro positivo: ");
    scanf("%d", &n);

    if (n <= 1) {
        primo = 0; // 0 e 1 não são primos
    } else {
        for (i = 2; i <= n / 2; i++) {
            if (n % i == 0) {
                primo = 0;
                break;
            }
        }
    }

    if (primo)
        printf("%d e primo.\n", n);
    else
        printf("%d nao e primo.\n", n);

    return 0;
}
```

6.4.

```
#include <stdio.h>

int main() {
    int n, i;
    unsigned long long fatorial = 1;

    printf("Digite um numero inteiro nao negativo: ");
    scanf("%d", &n);

    if (n < 0) {
        printf("Apenas numeros negativos.\n");
    } else {
        for (i = 1; i <= n; i++) {
            fatorial *= i;
        }
        printf("%d! = %llu\n", n, fatorial);
    }

    return 0;
}
```

6.5.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int secreto, palpite;

    srand(time(NULL)); // inicia gerador com hora atual
    secreto = rand() % 100 + 1; // número entre 1 e 100

    printf("Adivinhe o numero secreto (1 a 100):\n");

    do {
        printf("Palpite: ");
        scanf("%d", &palpite);
```

```
        if (palpite > secreto) {
            printf("Muito alto! Tentes um numero
                    menor.\n");
        } else if (palpite < secreto) {
            printf("Muito baixo! Tentes um numero
                    maior.\n");
        } else {
            printf("Parabens! Adivinhaste numero
                   secreto.\n");
        }

    } while (palpite != secreto);

    return 0;
}
```

6.6.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    char direcao;
    char saida;
    int tentativas = 0;

    // Inicializa o gerador de números aleatórios
    srand(time(NULL));

    // Define aleatoriamente a direção da saída
    switch (rand() % 4) {
        case 0: saida = 'N'; break;
        case 1: saida = 'S'; break;
        case 2: saida = 'L'; break;
        case 3: saida = 'O'; break;

    }

    printf("Estas preso num labirinto misterioso\n");
    printf("Tentes encontrar a saida!\n\n");
```

```
    do {
        printf("Digites uma direcao (N/S/L/O): ");
        scanf(" %c", &direcao);
        if (direcao == 'n') { direcao = 'N'; }
        else if (direcao == 's') { direcao = 'S'; }
        else if (direcao == 'l') { direcao = 'L'; }
        else if (direcao == 'o') { direcao = 'O'; }
        tentativas++;

        switch (direcao) {
            case 'N':
            case 'S':
            case 'L':
            case 'O':
                if (direcao == saida) {
                    printf("Encontraste a saida em %d
                         tentativa(s)!\n", tentativas);
                } else {
                    printf("Parede! Tentes outra
                           direcao.\n");
                }
                break;

            default:
                printf("Direcao invalida. Use N, S, L
                       ou O.\n");
        }

    } while (direcao != saida);

    return 0;
}
```

6.7.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
```

```
int dadosJogador, dado1, dado2, totalJogador,
     totalComputador;
char continuar;

// Inicializa o gerador de números aleatórios
srand(time(NULL));

printf("Bem-vindo ao jogo de dados!\n");

do {
    // Escolha do número de dados
    do {
        printf("\nQuantos dados jogar? (1 ou 2):");
        scanf("%d", &dadosJogador);
    } while (dadosJogador !=1 && dadosJogador !=2);

    // Jogada do jogador
    if (dadosJogador == 1) {
        dado1 = rand() % 6 + 1;
        printf("Jogador: %d\n", dado1);
        totalJogador = dado1;
    } else {
        dado1 = rand() % 6 + 1;
        dado2 = rand() % 6 + 1;
        printf("Jogador: %d e %d (Total: %d)\n",
               dado1, dado2, dado1 + dado2);
        totalJogador = dado1 + dado2;
    }

    /* Jogada do computador (sempre com o mesmo
       número de dados)*/

    if (dadosJogador == 1) {
        dado1 = rand() % 6 + 1;
        printf("Computador: %d\n", dado1);
        totalComputador = dado1;

    } else {
        dado1 = rand() % 6 + 1;
        dado2 = rand() % 6 + 1;
        printf("Computador: %d e %d (Total: %d)\n",
               dado1, dado2, dado1 + dado2);
```

```
            totalComputador = dado1 + dado2;
        }

        // Comparação dos resultados
        if (totalJogador > totalComputador) {
            printf("Venceste esta ronda!\n");

        } else if (totalJogador < totalComputador) {
            printf("Computador venceu esta ronda.\n");

        } else {
            printf("Empate!\n");
        }

        // Pergunta para continuar
        printf("\nDesejas jogar novamente? (s/n): ");
        scanf(" %c", &continuar);

    } while (continuar == 's' || continuar == 'S');

    printf("Obrigado por jogar!\n");

    return 0;
}
```

6.8.

```
#include <stdio.h>

int main() {
    int senhaCorreta = 1234;
    int senhaDigitada;
    int tentativas = 3;

    while (tentativas > 0) {
        printf("Digite a senha: ");
        scanf("%d", &senhaDigitada);

        if (senhaDigitada == senhaCorreta) {
            printf("Bem-vindo!\n");
            break;
```

```
        } else {
            tentativas--;
            if (tentativas > 0) {
                printf("Senha incorreta.
                        Tentativas restantes: %d\n",
                        tentativas);

            } else {
                printf("Acesso bloqueado.\n");
            }
        }
    }

    return 0;
}
```

15.4 Capítulo 7

7.1.

```
#include <stdio.h>

int main() {
    int v[10], i, soma = 0;

    for (i = 0; i < 10; i++) {
        printf("Digitar o %do numero: ", i + 1);
        scanf("%d", &v[i]);
        soma += v[i];
    }

    printf("Soma de todos os elementos = %d\n", soma);
    return 0;
}
```

7.2.

```
#include <stdio.h>

int main() {
    int v[10], i, pares = 0;

    for (i = 0; i < 10; i++) {
        printf("Digitar %d numero inteiro: ", i + 1);
        scanf("%d", &v[i]);
    }

    printf("\nValores introduzidos: ");
    for (i = 0; i < 10; i++) {
        printf("%d ", v[i]);
        if (v[i] % 2 == 0) {
            pares++;
        }
```

```
    }

    printf("\nTotal de numeros pares = %d\n", pares);
    return 0;
}
```

7.3.

```
#include <stdio.h>

int main() {
    int m[3][3], i, j;

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("Elemento [%d][%d]: ", i, j);
            scanf("%d", &m[i][j]);
        }
    }

    printf("\nDiagonal principal: ");
    for (i = 0; i < 3; i++) {
        printf("%d ", m[i][i]);
    }
    printf("\n");

    return 0;
}
```

7.4.

```
#include <stdio.h>

int main() {
    float m[3][3];
    float somaLinha, somaColuna;
    int i, j;

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("Elemento [%d][%d]: ", i, j);
            scanf("%f", &m[i][j]);
```

```
        }
    }

    for (i = 0; i < 3; i++) {
        somaLinha = 0;
        for (j = 0; j < 3; j++) {
            somaLinha += m[i][j];
        }
        printf("Soma da linha %d = %.2f\n", i,
                somaLinha);
    }

    for (j = 0; j < 3; j++) {
        somaColuna = 0;
        for (i = 0; i < 3; i++) {
            somaColuna += m[i][j];
        }
        printf("Soma da coluna %d = %.2f\n", j,
               somaColuna);
    }

    return 0;
}
```

7.5.

```
#include<stdio.h>
int main() {
    int matriz[6][6];
    int i, j;

    for (i = 0; i < 6; i++) {
        for (j = 0; j < 6; j++) {
            if (i == 0 || i == 5 || j == 0 || j == 5) {
                matriz[i][j] = 1;  // borda
            } else {
                matriz[i][j] = 0;  // interior
            }
        }
    }
```

```
    // Imprimir a matriz
    for (i = 0; i < 6; i++) {
        for (j = 0; j < 6; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

7.6.

```
#include <stdio.h>

int main() {
    float A[3][3], B[3][3];
    int i, j;

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("A[%d][%d]: ", i, j);
            scanf("%f", &A[i][j]);
        }
    }
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            B[i][j] = A[i][j] - A[i][i];
        }
    }

    printf("\nMatriz B:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("%6.2f ", B[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

7.7.

```
#include <stdio.h>

int main() {
    int i, j;
    for (i = 0; i < 6; i++) {
        int valor = (i % 3) + 1; // 1,2,3,1,2,3
        for (j = 0; j < 6; j++) {
            printf("%d", valor);
            if (j < 5) printf("\t");
        }
        printf("\n");
    }
    return 0;
}
```

7.8.

```
#include <stdio.h>

int main() {
    char m[8][8];
    int i, j;
    int nP=0, nT=0, nC=0, nB=0, nR=0, nD=0, nVazias=0;

    printf("Introduza o tabuleiro 8x8 (caracteres:
            P,T,C,B,R,D,-):\n");
    for (i = 0; i < 8; i++) {
        for (j = 0; j < 8; j++) {
            char ch;
 /* espaço antes de %c ignora newlines/espacos
pendentes */
            if (scanf(" %c", &ch) != 1) return 0;

            // valida e conta
            if (ch=='P') nP++;
            else if (ch=='T') nT++;
            else if (ch=='C') nC++;
            else if (ch=='B') nB++;
            else if (ch=='R') nR++;
            else if (ch=='D') nD++;
```

```
            else if (ch=='-') nVazias++;
            else {
                printf("Caractere invalido. Use apenas
                       P,T,C,B,R,D,-.\n");
                j--; // repete esta coluna
                continue;
            }
            m[i][j] = ch;
        }
    }

    printf("\nContagem de pecas:\n");
    printf("Peoes (P): %d\n", nP);
    printf("Torres (T): %d\n", nT);
    printf("Cavalos (C): %d\n", nC);
    printf("Bispos (B): %d\n", nB);
    printf("Reis (R): %d\n", nR);
    printf("Damas (D): %d\n", nD);
    printf("Vazias (-): %d\n", nVazias);
    return 0;
}
```

7.9.

```
#include <stdio.h>

int main() {
    char mapa[10][10];
    int i, j;

    // preenche com '.'
    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)
            mapa[i][j] = '.';

    printf("Introduza 5 obstaculos (linha e coluna
            entre 0 e 9):\n");
    for (int k = 1; k <= 5; k++) {
        int lin, col;
        while (1) {
            printf("Obstaculo %d - linha coluna: ", k);
```

```
            if (scanf("%d %d", &lin, &col) != 2)
               return 0;
            if (lin <0 || lin >9 || col <0 || col >9) {
                printf("Posicao fora dos limites. Tente
                     de novo.\n");
            } else if (mapa[lin][col] == 'X') {
                printf("Ja existe obstaculo nessa
                      posicao. Tente outra.\n");
            } else {
                mapa[lin][col] = 'X';
                break;
            }
        }
    }

    // imprime o mapa
    printf("\nMapa 10x10:\n");
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 10; j++) {
            printf("%c ", mapa[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

7.10.

```
#include<stdio.h>
#include<stdlib.h>
int main() {

    char matriz[5][5];
    int i, j;
    int linhaT, colunaT;
    int linhaJ, colunaJ;

    // Inicializar a matriz com '-'
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 5; j++) {
            matriz[i][j] = '-';
```

```
        }
    }

     do {
        printf("Introduza a posição do tesouro (linha e
               coluna entre 0 e 4): ");
        scanf("%d %d", &linhaT, &colunaT);

        if (linhaT < 0 || linhaT > 4 || colunaT < 0 ||
            colunaT > 4) {
            printf("Erro: fora dos limites.\n");
        }
    } while (linhaT < 0 || linhaT > 4 || colunaT < 0 ||
            colunaT > 4);

      /* Ler posição do jogador com verificação de
limites e sobreposição*/
    do {
        printf("Introduza a posição do jogador (linha e
                coluna entre 0 e 4): ");
        scanf("%d %d", &linhaJ, &colunaJ);

        if (linhaJ < 0 || linhaJ > 4 || colunaJ < 0 ||
            colunaJ > 4) {
            printf("Erro: fora dos limites.\n");
        } else if (linhaJ == linhaT && colunaJ ==
                  colunaT) {
            printf("Erro: o jogador não pode estar na
                   mesma posição do tesouro.\n");
        }
    } while (
        linhaJ < 0 || linhaJ > 4 ||
        colunaJ < 0 || colunaJ > 4 ||
        (linhaJ == linhaT && colunaJ == colunaT)
    );

    // Marcar posições
    matriz[linhaT][colunaT] = 'T';
    matriz[linhaJ][colunaJ] = 'J';

    // Imprimir o tabuleiro
    printf("\nTabuleiro:\n");
```

```
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 5; j++) {
            printf("%c ", matriz[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

7.11.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int matriz[7][7];
    int i, j;

    for (i = 0; i < 7; i++) {
        for (j = 0; j < 7; j++) {
            if (i == 3 || j == 3) {
                matriz[i][j] = 1;
             // linha central ou coluna central
            } else {
                matriz[i][j] = 0;
            }
        }
    }

    // Imprimir a matriz
    for (i = 0; i < 7; i++) {
        for (j = 0; j < 7; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

7.12.

```
#include<stdio.h>
int main() {
    int matriz[6][6];
    int i, j;

    for (i = 0; i < 6; i++) {
        for (j = 0; j < 6; j++) {
            if (i == 0 || i == 5 || j == 0 || j == 5) {
                matriz[i][j] = 1;  // borda
            } else {
                matriz[i][j] = 0;  // interior
            }
        }
    }

    // Imprimir a matriz
    for (i = 0; i < 6; i++) {
        for (j = 0; j < 6; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

15.5 Capítulo 8

8.1.

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[101];
    int i, cont = 0;

    printf("Introduza uma string: ");
    fgets(str, sizeof(str), stdin);

    for (i = 0; str[i] != '\0'; i++) {
        char c = str[i];
        if (c=='a'||c=='A'||c=='e'||c=='E'||c=='i'||
           c=='I'||c=='o'||c=='O'||c=='u'||c=='U') {
            cont++;
        }
    }

    printf("Total de vogais: %d\n", cont);
    return 0;
}
```

8.2.

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[81], inv[81];
    int i, j, len;

    printf("Introduza a string: ");
    fgets(str, sizeof(str), stdin);
```

```
    len = strlen(str);
    if (str[len-1] == '\n') str[len-1] = '\0';
    len = strlen(str);

    // copia espaços
    for (i = 0; i < len; i++) {
        if (str[i] == ' ')
            inv[i] = ' ';
        else
            inv[i] = '\0';
    }

    // preenche invertendo apenas caracteres não-espaço
    for (i = 0, j = len - 1; i < len; i++) {
        if (inv[i] == ' ') continue;
        while (j >= 0 && str[j] == ' ') j--;
        inv[i] = str[j];
        j--;
    }

    inv[len] = '\0';
    printf("Invertida mantendo espacos: %s\n", inv);
    return 0;
}
```

8.3.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main() {
    char str[81], dig[81];
    int i, j = 0;

    printf("Introduza a string: ");
    fgets(str, sizeof(str), stdin);

    for (i = 0; str[i] != '\0'; i++) {
        if (isdigit((unsigned char)str[i])) {
            dig[j++] = str[i];
        }
```

```
    }
    dig[j] = '\0';

    printf("String original: %s", str);
    printf("So digitos: %s\n", dig);
    return 0;
}
```

8.4.

```
#include <stdio.h>
#include <string.h>

int main() {
    char palavra[51];
    int i, j, ehCapicua = 1;

    printf("Introduza a palavra: ");
    scanf("%50s", palavra);

    j = strlen(palavra) - 1;
    for (i = 0; i < j; i++, j--) {
        if (palavra[i] != palavra[j]) {
            ehCapicua = 0;
            break;
        }
    }

    if (ehCapicua)
        printf("E capicua.\n");
    else
        printf("Nao e capicua.\n");

    return 0;
}
```

8.5.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
```

```
int main() {
    char frase[101];
    int i, palavras = 0, dentroPalavra = 0;

    printf("Introduza a frase: ");
    fgets(frase, sizeof(frase), stdin);

    for (i = 0; frase[i] != '\0'; i++) {
        if (!isspace((unsigned char)frase[i]) &&
             dentroPalavra == 0) {
            dentroPalavra = 1;
            palavras++;
        } else if (isspace((unsigned char)frase[i])) {
            dentroPalavra = 0;
        }
    }

    printf("Total de palavras: %d\n", palavras);
    return 0;
}
```

8.6.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main() {
    char frase[121];
    int i;

    printf("Introduza a frase: ");
    fgets(frase, sizeof(frase), stdin);

    for (i = 0; frase[i] != '\0'; i++) {
        if (isdigit((unsigned char)frase[i])) {
            frase[i] = '*';
        }
    }

    printf("Frase com numeros ocultos: %s", frase);
```

```
    return 0;
}
```

8.7.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main() {
    char p1[31], p2[31];
    int iguais = 1, i;

    printf("Primeira palavra: ");
    scanf("%30s", p1);
    printf("Segunda palavra: ");
    scanf("%30s", p2);

    if (strlen(p1) != strlen(p2)) {
        iguais = 0;
    } else {
        for (i = 0; p1[i] != '\0'; i++) {
            if (tolower((unsigned char)p1[i]) !=
               tolower((unsigned char)p2[i])) {
                iguais = 0;
                break;
            }
        }
    }

    if (iguais)
        printf("Iguais: %s\n", p1);
    else
        printf("Diferentes: %s e %s\n", p1, p2);

    return 0;
}
```

8.8.

```
#include <stdio.h>
#include<stdlib.h>
```

```
#include <string.h>
#include <ctype.h>

int main() {
    char secreta[21] = "TESTE";
// Palavra secreta (máx. 20 caracteres)
    char estado[21];
    char letra;
    int tentativas = 10;
    int tamanho = strlen(secreta);
    int acertos = 0;
    int i, encontrado;

    // Inicializa estado com '_'
    for (i = 0; i < tamanho; i++) {
        estado[i] = '_';
    }
    estado[tamanho] = '\0';

    printf("=== Mini Jogo da Forca ===\n");
    printf("Adivinhe a palavra! Tens %d tentativas.
           \n\n", tentativas);

    while (tentativas > 0 && acertos < tamanho) {
        printf("Palavra: %s\n", estado);
        printf("Tentativas restantes: %d\n",
                tentativas);
        printf("Digite uma letra: ");
        scanf(" %c", &letra);
        letra = toupper(letra); // Garantir maiúscula

        encontrado = 0;
        for (i = 0; i < tamanho; i++) {
            if (secreta[i] == letra && estado[i] ==
'_') {
                estado[i] = letra;
                acertos++;
                encontrado = 1;
            }
        }

        if (!encontrado) {
```

```
            printf("Letra '%c' nao encontrada!\n",
letra);
            tentativas--;
        }
        printf("\n");
    }

    if (acertos == tamanho) {
        printf("Parabens! Acertastes: %s\n", secreta);
    } else {
        printf("Perdeste! Palavra era: %s\n", secreta);
    }

    return 0;
}
```

8.9.

```
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
#include <ctype.h>


int main() {
    char frase[81];
    int i;

    printf("Digite uma frase: ");
    fgets(frase, sizeof(frase), stdin);

    // Remove \n final, se existir
    frase[strcspn(frase, "\n")] = '\0';

    for (i = 0; frase[i] != '\0'; i++) {
        if (isalpha(frase[i])) {
           char base = isupper(frase[i]) ? 'A' : 'a';
           frase[i]=((frase[i] - base + 3) % 26 )+base;
        }
    }

    printf("Frase codificada: %s\n", frase);
```

```
    return 0;
}
```

8.10.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>

#define TAM 10

int main() {
    char tabuleiro[TAM][TAM];
    char palavra[11];
    int i, j;
    int linha, coluna;
    int tamanho;

    srand(time(NULL));

// Preencher tabuleiro com letras maiúsculas aleatórias
    for (i = 0; i < TAM; i++) {
        for (j = 0; j < TAM; j++) {
            tabuleiro[i][j] = 'A' + rand() % 26;
        }
    }

    // Ler palavra
    printf("Digite uma palavra (max. 10 letras): ");
    fgets(palavra, sizeof(palavra), stdin);
    palavra[strcspn(palavra, "\n")] = '\0';

    // Garantir que está em maiúsculas
    for (i = 0; palavra[i] != '\0'; i++) {
        palavra[i] = toupper(palavra[i]);
    }

    tamanho = strlen(palavra);
    if (tamanho == 0 || tamanho > TAM) {
        printf("Tamanho da palavra inválido.\n");
```

```
            return 1;
        }

        // Escolher posição aleatória válida
        linha = rand() % TAM;
        coluna = rand() % (TAM - tamanho + 1);

        // Inserir palavra no tabuleiro
        for (i = 0; i < tamanho; i++) {
            tabuleiro[linha][coluna + i] = palavra[i];
        }

        // Mostrar tabuleiro
        printf("\nTabuleiro:\n");
        for (i = 0; i < TAM; i++) {
            for (j = 0; j < TAM; j++) {
                printf("%c ", tabuleiro[i][j]);
            }
            printf("\n");
        }

        // Mostrar posição inicial
        printf("\nPalavra inserida na linha %d, coluna
                %d.\n", linha, coluna);

        return 0;
}
```

15.6 Capítulo 9

9.1.

```
#include <stdio.h>
#include <stdio.h>

void incrementa(int x) {
    x = x + 5;
    printf("Dentro da funcao, contador = %d\n", x);
}

int main() {
    int contador = 10;

    incrementa(contador);
    printf("No main, contador = %d\n", contador);

    return 0;
}
```

9.2.

```
#include <stdio.h>

int numero = 100; // variável global

void mostraNumero() {
    printf("Variavel global numero = %d\n", numero);
}

int main() {
    int numero = 50; // variável local

    printf("Variavel local numero = %d\n", numero);
    mostraNumero();
```

```
    return 0;
}
```

9.3.

```
#include <stdio.h>

int dano(int energia, int valor) {
    if (valor <= 0) {
        printf("Invalido: digitar numero positivo.\n");
        return energia; // sem alteracao
    }
    energia -= valor;
    if (energia < 0) energia = 0;
    return energia;
}

int curar(int energia, int valor) {
    if (valor <= 0) {
        printf("Invalido: digitar numero positivo.\n");
        return energia; // sem alteracao
    }
    energia += valor;
    if (energia > 100) energia = 100;
    return energia;
}

void mostrarEnergia(int energia) {
    printf("Energia atual: %d\n", energia);
}

int main() {
    int energia = 100;
    int opcao, valor;

    do {
        printf("\n--- Menu ---\n");
        printf("1. Sofrer dano\n");
        printf("2. Recuperar energia\n");
        printf("3. Mostrar energia\n");
        printf("4. Sair\n");
        printf("Escolha: ");
```

```
 if (scanf("%d", &opcao) != 1) {
// leitura invalida: limpar stdin e continuar
     int c;
     while ((c = getchar()) != '\n' && c !=
            EOF);
     printf("Opcao invalida.\n");
     continue;
 }

 switch (opcao) {
     case 1:
         printf("Quanto de dano sofreu? ");
         if (scanf("%d", &valor) != 1) {
             int c;
             while ((c = getchar()) != '\n' &&
                     c != EOF);
             printf("Entrada invalida.\n");
             break;
         }
         energia = dano(energia, valor);
         mostrarEnergia(energia);
         break;

     case 2:
         printf("Quanto deseja curar? ");
         if (scanf("%d", &valor) != 1) {
             int c;
             while ((c = getchar()) != '\n' &&
                     c != EOF);
             printf("Entrada invalida.\n");
             break;
         }
         energia = curar(energia, valor);
         mostrarEnergia(energia);
         break;

     case 3:
         mostrarEnergia(energia);
         break;

     case 4:
         printf("A terminar...\n");
```

```
                break;

            default:
                printf("Opcao invalida.\n");
        }

        if (energia == 0) {
           printf("\nZero de Energia. Fim de jogo.\n");
            break;
        }
    } while (opcao != 4);



    system("pause>NULL");
    return 0;
}
```

15.7 Capítulo 10

10.1.

```
#include <stdio.h>

int main() {
    int a, b, soma;
    int *pa = &a, *pb = &b;

    printf("Introduza o primeiro numero: ");
    scanf("%d", pa);
    printf("Introduza o segundo numero: ");
    scanf("%d", pb);

    soma = *pa + *pb;

    printf("Primeiro numero: %d\n", *pa);
    printf("Segundo numero: %d\n", *pb);
    printf("Soma (via apontadores): %d\n", soma);

    return 0;
}
```

10.2.

```
#include <stdio.h>

int main() {
    int v[6], inv[6];
    int *pOrig = v, *pInv = inv;
    int i;

    for (i = 0; i < 6; i++) {
        printf("Introduza o elemento %d: ", i + 1);
        scanf("%d", pOrig + i);
    }
```

```
    // inverter
    for (i = 0; i < 6; i++) {
        *(pInv + i) = *(pOrig + (5 - i));
    }

    printf("Vetor original: ");
    for (i = 0; i < 6; i++) {
        printf("%d ", *(pOrig + i));
    }

    printf("\nVetor invertido: ");
    for (i = 0; i < 6; i++) {
        printf("%d ", *(pInv + i));
    }

    printf("\n");
    return 0;
}
```

10.3.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, i;
    int *v;
    int maior, menor;
    float soma = 0;

    printf("Tamanho do vetor: ");
    scanf("%d", &n);

    v = (int*) malloc(n * sizeof(int));
    if (v == NULL) {
        printf("Erro ao alocar memoria.\n");
        return 1;
    }

    for (i = 0; i < n; i++) {
        printf("Elemento %d: ", i + 1);
        scanf("%d", &v[i]);
```

```
        soma += v[i];
    }

    maior = menor = v[0];
    for (i = 1; i < n; i++) {
        if (v[i] > maior) maior = v[i];
        if (v[i] < menor) menor = v[i];
    }

    printf("Maior: %d\n", maior);
    printf("Menor: %d\n", menor);
    printf("Media: %.2f\n", soma / n);

    free(v);
    return 0;
}
```

10.4.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *v;
    int i, soma = 0;

    v = (int*) malloc(5 * sizeof(int));
    if (v == NULL) {
        printf("Erro ao alocar memoria.\n");
        return 1;
    }

    for (i = 0; i < 5; i++) {
        printf("Elemento %d: ", i + 1);
        scanf("%d", &v[i]);
    }

    // redimensionar para 10
    v = (int*) realloc(v, 10 * sizeof(int));
    if (v == NULL) {
        printf("Erro ao realocar memoria.\n");
        return 1;
```

```
    }

    for (i = 5; i < 10; i++) {
        printf("Elemento %d: ", i + 1);
        scanf("%d", &v[i]);
    }

    printf("Vetor apos expansao: ");
    for (i = 0; i < 10; i++) {
        printf("%d ", v[i]);
        soma += v[i];
    }

    printf("\nSoma total: %d\n", soma);

    free(v);
    return 0;
}
```

10.5.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *v;
    int i, soma = 0;

    v = (int*) calloc(100, sizeof(int));
    if (v == NULL) {
        printf("Erro ao alocar memoria.\n");
        return 1;
    }

    // verificar inicialização
    for (i = 0; i < 10; i++) {
        printf("%d ", v[i]); // deve imprimir 0
    }
    printf("... (todos inicializados com zero)\n");

    // preencher de 1 a 100
```

```
    for (i = 0; i < 100; i++) {
        v[i] = i + 1;
        soma += v[i];
    }

    printf("Primeiros 10 valores: ");
    for (i = 0; i < 10; i++) {
        printf("%d ", v[i]);
    }
    printf("\nSoma total: %d\n", soma);

    free(v);
    return 0;
}
```

15.8 Capítulo 11

11.1.

```
#include <stdio.h>

void transformarPoder(int *energia1, int *energia2) {
    *energia1 = *energia1 / 2;
    *energia2 = *energia2 / 2;
}
int main() {
    int p1 = 100, p2 = 100;

    printf("Antes do ataque especial:\n");
    printf("Jogador 1: %d\n", p1);
    printf("Jogador 2: %d\n", p2);

    transformarPoder(&p1, &p2);

    printf("\nDepois do ataque especial:\n");
    printf("Jogador 1: %d\n", p1);
    printf("Jogador 2: %d\n", p2);
    return 0;
}
```

11.2.

```
#include <stdio.h>

void batalhaNumeros(int n1, int n2, int *maior, int
*menor) {
    if (n1 > n2) {
        *maior = n1;
        *menor = n2;
    } else {
        *maior = n2;
        *menor = n1;
    }
}
```

```
int main() {
    int num1, num2, campeao, desafiante;

    printf("Jogador 1, introduza o numero: ");
    scanf("%d", &num1);
    printf("Jogador 2, introduza o numero: ");
    scanf("%d", &num2);
    batalhaNumeros(num1, num2, &campeao, &desafiante);
    printf("\nCampeao: %d\n", campeao);
    printf("Desafiante: %d\n", desafiante);
    return 0;
}
```

11.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void gerarLoot(int *vet, int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        vet[i] = rand() % 101; // 0 a 100
    }
}

int main() {
    int n, i;

    printf("Quantos itens no bau? ");
    scanf("%d", &n);

    int itens[n];
    srand(time(NULL)); // inicializa aleatório
    gerarLoot(itens, n);
    printf("\nItens no bau:\n");
    for (i = 0; i < n; i++) {
       printf("Item %d: %d pontos\n", i + 1, itens[i]);
    }

    return 0;
}
```

15.9 Capítulo 12

12.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TAM_NOME 50

typedef struct {
    char nome[TAM_NOME];
    int idade;
    float altura;
} Pessoa;

void removerNovaLinha(char *str) {
    str[strcspn(str, "\n")] = '\0';
}

// Função para ler dados com validação (por referência)
void lerPessoa(Pessoa *p) {
    printf("Nome: ");
    fgets(p->nome, TAM_NOME, stdin);
    removerNovaLinha(p->nome);

    do {
        printf("Idade (5 a 12 anos): ");
        scanf("%d", &p->idade);
        if (p->idade < 5 || p->idade > 12) {
            printf("Idade inválida. Digite de novo\n");
        }
    } while (p->idade < 5 || p->idade > 12);

    do {
        printf("Altura (1.0 m a 1.5 m): ");
        scanf("%f", &p->altura);
        if (p->altura < 1.0 || p->altura > 1.5) {
```

```
            printf("Altura inválida. Digite de
novo\n");
        }
    } while (p->altura < 1.0 || p->altura > 1.5);
    getchar(); // Limpa o '\n' do buffer
}

// Função para exibir dados (por valor)
void imprimirPessoa(Pessoa p) {
    printf("Nome: %-20s | Idade: %2d | Altura: %.2f
            m\n",p.nome, p.idade, p.altura);
}

int main() {
    int n;

    printf("Quantas crianças deseja registar? ");
    scanf("%d", &n);
    getchar(); // Limpa buffer

    Pessoa *criancas = (Pessoa *)
                       malloc(n * sizeof(Pessoa));
    if (criancas == NULL) {
        printf("Erro: impossível alocar memória.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        printf("\n--- Criança %d ---\n", i + 1);
        lerPessoa(&criancas[i]);
    }

    printf("\n===== Lista de Crianças =====\n");
    for (int i = 0; i < n; i++) {
        imprimirPessoa(criancas[i]);
    }

    free(criancas);

    return 0;
}
```

12.2.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define TAM_NOME 50

typedef struct {
    char nome[TAM_NOME];
    int  idade;      // [5..12]
    float altura;    // [1.0..1.5]
} Pessoa;

/* Protótipos (valor vs. referência) */
void  removerNovaLinha(char *s);
void  limparBufferEntrada(void);

void  lerPessoa(Pessoa *p);  // referência (altera p)
void  imprimirPessoa(Pessoa p); // valor (não altera p)
float mediaIdades(Pessoa v[], int n);
// não altera o vetor
Pessoa maiorAltura(Pessoa v[], int n);
// devolve por valor

int compararIgnoreCase(const char *a, const char *b);
int procurarPessoa(Pessoa v[], int n,const char *nome);
     // devolve índice ou -1
int   contarAcimaMediaAltura(Pessoa v[], int n);
     // não altera
void  ordenarPorAltura(Pessoa v[], int n);
     // referência (altera vetor)
void  alterarPessoa(Pessoa *p);
     // referência (altera p)

/* Implementação */
void removerNovaLinha(char *s) {
    s[strcspn(s, "\n")] = '\0';
}

void limparBufferEntrada(void) {
    int c;
```

```
    while ((c = getchar()) != '\n' && c != EOF) { }
}

void lerPessoa(Pessoa *p) {
    printf("Nome: ");
    fgets(p->nome, TAM_NOME, stdin);
    removerNovaLinha(p->nome);

    do {
        printf("Idade (5 a 12): ");
        if (scanf("%d", &p->idade) != 1) {
            limparBufferEntrada(); p->idade = -1; }
        if (p->idade < 5 || p->idade > 12)
            printf("Valor invalido.\n");
    } while (p->idade < 5 || p->idade > 12);

    do {
        printf("Altura (1.0 a 1.5 m): ");
        if (scanf("%f", &p->altura) != 1) {
limparBufferEntrada(); p->altura = -1.0f; }
        if (p->altura < 1.0f || p->altura > 1.5f)
            printf("Valor invalido.\n");
    } while (p->altura < 1.0f || p->altura > 1.5f);

    limparBufferEntrada();
}

void imprimirPessoa(Pessoa p) {
    printf("Nome: %-20s | Idade: %2d | Altura: %.2f
           m\n", p.nome, p.idade, p.altura);
}

float mediaIdades(Pessoa v[], int n) {
    int soma = 0;
    for (int i = 0; i < n; i++) soma += v[i].idade;
    return (n > 0) ? (float)soma / n : 0.0f;
}

Pessoa maiorAltura(Pessoa v[], int n) {
    Pessoa max = v[0];
    for (int i = 1; i < n; i++) {
        if (v[i].altura > max.altura) max = v[i];
```

```
    }
    return max; // retorno por valor
}
int compararIgnoreCase(const char *a, const char *b) {
    while (*a && *b) {
        char ca = (char)tolower((unsigned char)*a);
        char cb = (char)tolower((unsigned char)*b);
        if (ca != cb)
          return (unsigned char)ca - (unsigned char)cb;
        a++; b++;
    }
    return (unsigned char)*a - (unsigned char)*b;
}

int procurarPessoa(Pessoa v[], int n, const char *nome)
{
    for (int i = 0; i < n; i++)
        if (compararIgnoreCase(v[i].nome, nome) == 0)
             return i;
    return -1;
}

int contarAcimaMediaAltura(Pessoa v[], int n) {
    float soma = 0.0f;
    for (int i = 0; i < n; i++) soma += v[i].altura;
    float media = (n > 0) ? soma / n : 0.0f;

    int cont = 0;
    for (int i = 0; i < n; i++)
        if (v[i].altura > media) cont++;
    return cont;
}

void ordenarPorAltura(Pessoa v[], int n) {
    // insertion sort (estável, simples para ensino)
    for (int i = 1; i < n; i++) {
        Pessoa chave = v[i];
        int j = i - 1;
        while (j >= 0 && v[j].altura > chave.altura) {
            v[j+1] = v[j];
            j--;
        }
```

```
        v[j+1] = chave;
    }
}

void alterarPessoa(Pessoa *p) {
    printf("A alterar registo de: %s\n", p->nome);
    lerPessoa(p);
// reaproveita validação; altera por referência
}

int main(void) {
    int n;

    printf("Quantidade de alunos/criancas: ");
    if (scanf("%d", &n) != 1 || n <= 0) {
        printf("Valor invalido.\n");
        return 1;
    }
    limparBufferEntrada();

    Pessoa *turma = (Pessoa*)
                    malloc(n * sizeof(Pessoa));
    if (!turma) {
        printf("Falha de memoria.\n");
        return 1; }

    for (int i = 0; i < n; i++) {
        printf("\n--- Registo %d ---\n", i+1);
        lerPessoa(&turma[i]); // por referência
    }

    int opc;
    do {
        printf("\n===== Menu 12.1 =====\n");
        printf("1) Listar todos\n");
        printf("2) Media das idades\n");
        printf("3) Pessoa mais alta\n");
        printf("4) Procurar por nome\n");
        printf("5) Contar acima da media de altura\n");
        printf("6) Ordenar por altura (asc)\n");
        printf("7) Alterar registo por indice\n");
        printf("8) Sair\n");
```

```
printf("Opcao: ");
if (scanf("%d", &opc) != 1) {
    limparBufferEntrada(); opc = 0; }

if (opc == 1) {
    printf("\n--- Lista ---\n");
    for (int i = 0; i < n; i++)
       imprimirPessoa(turma[i]); // por valor
}
else if (opc == 2) {
    float m = mediaIdades(turma, n);
    printf("Media das idades: %.2f\n", m);
}
else if (opc == 3) {
    Pessoa p = maiorAltura(turma, n);
       // retorna por valor
    printf("Mais alto(a): ");
    imprimirPessoa(p);
}
else if (opc == 4) {
    char busca[TAM_NOME];
    limparBufferEntrada();
    printf("Nome a procurar: ");
    fgets(busca, TAM_NOME, stdin);
    removerNovaLinha(busca);
    int idx = procurarPessoa(turma, n, busca);
    if (idx >= 0) imprimirPessoa(turma[idx]);
    else printf("Nao encontrado.\n");
}
else if (opc == 5) {
    int k = contarAcimaMediaAltura(turma, n);
    printf("%d aluno(s) acima da media\n", k);
}
else if (opc == 6) {
    ordenarPorAltura(turma, n);
    // altera vetor (referência)
    printf("Ordem ascendente por altura\n");
}
else if (opc == 7) {
    int idx;
    printf("Indice [0..%d]: ", n-1);
```

```
            if (scanf("%d", &idx) == 1 && idx >= 0 &&
                idx < n) {
                limparBufferEntrada();
                alterarPessoa(&turma[idx]);
            } else {
                limparBufferEntrada();
                printf("Indice invalido.\n");
            }
        }
        else if (opc == 8) {
            printf("A terminar...\n");
        }
        else {
            printf("Opcao invalida\n");
        }

    } while (opc != 8);

    free(turma);

    return 0;
}
```

12.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* ------------------ Tipos ------------------ */
typedef struct {
    int dia, mes, ano;
} Data;

typedef struct {
    char piloto[50];
    char pista[50];
    float melhorVolta;   // em segundos (deve ser > 0)
    Data dataProva;
} Corrida;
```

```
/* ------------------ Protótipos ------------------ */
void limparBufferEntrada(void);
void removerNovaLinha(char *s);
int  compararIgnoreCase(const char *a, const char *b);

int  dataValida(Data d);
void lerData(Data *d);

void lerCorrida(Corrida *c);
                // referência (altera c)
void imprimirCorrida(Corrida c);
           // valor (não altera c)
Corrida melhorTempo(Corrida v[], int n);
    // devolve por valor
int  procurarPorPiloto(Corrida v[], int n,
                       const char *nome);

void alterarTempo(Corrida v[], int n);
      // referência (altera vetor)
void listarTodas(Corrida v[], int n);
        // só consulta
void inserirTodas(Corrida v[], int n);
       // referência (altera vetor)

/* ------------------ Utilidades ------------------ */
void limparBufferEntrada(void) {
    int c;
    while ((c = getchar()) != '\n' && c != EOF) { }
}

void removerNovaLinha(char *s) {
    s[strcspn(s, "\n")] = '\0';
}

int compararIgnoreCase(const char *a, const char *b) {
    while (*a && *b) {
        char ca = (char)tolower((unsigned char)*a);
        char cb = (char)tolower((unsigned char)*b);
        if (ca != cb)
          return (unsigned char)ca - (unsigned char)cb;
        a++; b++;
```

```
    }
    return (unsigned char)*a - (unsigned char)*b;
}

/* -------------- Validação de Data ---------------- */
int dataValida(Data d) {
    if (d.ano < 2020 || d.mes < 1 || d.mes > 12
         || d.dia < 1) return 0;

    int diasMes[] =
{0,31,28,31,30,31,30,31,31,30,31,30,31};
    int bissexto = ( (d.ano%4==0 && d.ano%100!=0)
                  || (d.ano%400==0) );
    if (bissexto) diasMes[2] = 29;

    if (d.dia > diasMes[d.mes]) return 0;
    return 1;
}

void lerData(Data *d) {
    do {
        printf("Data da prova (DD MM AAAA): ");
        if (scanf("%d %d %d", &d->dia, &d->mes,
             &d->ano) != 3) {
            limparBufferEntrada();
            d->dia = d->mes = d->ano = -1;
        }
        if (!dataValida(*d)) {
            printf("Data invalida. Redigite\n");
        }
    } while (!dataValida(*d));
    limparBufferEntrada();
}

/* ------------- Operações de Corrida -------------- */
void lerCorrida(Corrida *c) {
    printf("Piloto (max 49 chars): ");
    fgets(c->piloto, sizeof(c->piloto), stdin);
    removerNovaLinha(c->piloto);

    printf("Pista  (max 49 chars): ");
    fgets(c->pista, sizeof(c->pista), stdin);
```

```
    removerNovaLinha(c->pista);

    do {
        printf("Melhor volta (segundos, > 0): ");
        if (scanf("%f", &c->melhorVolta) != 1) {
            limparBufferEntrada();
            c->melhorVolta = -1.0f;
        }
        if (c->melhorVolta <= 0.0f) {
            printf("Valor invalido. Redigite\n");
        }
    } while (c->melhorVolta <= 0.0f);
    limparBufferEntrada();

    lerData(&c->dataProva);
}

void imprimirCorrida(Corrida c) {
    printf("Piloto: %-20s | Pista: %-15s | Melhor
           volta: %7.2fs | Data: %02d/%02d/%04d\n",
           c.piloto, c.pista, c.melhorVolta,
           c.dataProva.dia, c.dataProva.mes,
           c.dataProva.ano);
}

Corrida melhorTempo(Corrida v[], int n) {
    Corrida best = v[0];
    for (int i = 1; i < n; i++) {
        if (v[i].melhorVolta < best.melhorVolta)
            best = v[i];
    }
    return best; // retorno por valor
}

int procurarPorPiloto(Corrida v[], int n,
                      const char *nome) {
    for (int i = 0; i < n; i++)
        if (compararIgnoreCase(v[i].piloto, nome) == 0)
            return i;
    return -1;
}
```

```
/* altera o vetor (referência) */
void alterarTempo(Corrida v[], int n) {
    char nome[50];
    printf("Nome do piloto para alterar tempo: ");
    fgets(nome, sizeof(nome), stdin);
    removerNovaLinha(nome);

    int idx = procurarPorPiloto(v, n, nome);
    if (idx < 0) {
        printf("Piloto nao encontrado.\n");
        return;
    }

    float novo;
    do {
        printf("Novo tempo (segundos, > 0): ");
        if (scanf("%f", &novo) != 1) {
            limparBufferEntrada();
            novo = -1.0f;
        }
        if (novo <= 0.0f) printf("Valor invalido.\n");
    } while (novo <= 0.0f);
    limparBufferEntrada();

    v[idx].melhorVolta = novo;
    printf("Tempo atualizado com sucesso.\n");
}

void listarTodas(Corrida v[], int n) {
    if (n == 0) {
        printf("Sem registos.\n");
        return; }
    for (int i = 0; i < n; i++) {
        printf("[%d] ", i);
        imprimirCorrida(v[i]);
    }
}

void inserirTodas(Corrida v[], int n) {
    for (int i = 0; i < n; i++) {
        printf("\n--- Corrida %d ---\n", i + 1);
        lerCorrida(&v[i]); // referência
```

```
    }
}

/* ------------------ Main (menu) ------------------ */
int main(void) {
    int n, opc;
    Corrida *corridas = NULL;

    printf("Quantidade de corridas a registar: ");
    if (scanf("%d", &n) != 1 || n <= 0) {
        printf("Valor invalido.\n");
        return 1;
    }
    limparBufferEntrada();

    corridas = (Corrida*) malloc(n * sizeof(Corrida));
    if (!corridas) {
        printf("Falha de memoria.\n");
        return 1;
    }

    int preenchidas = 0;

    do {
        printf("\n===== Campeonato de Corridas
=====\n");
        printf("1) Inserir/editar todas as
corridas\n");
        printf("2) Listar corridas\n");
        printf("3) Mostrar corrida com melhor
               tempo\n");
        printf("4) Alterar tempo de um piloto\n");
        printf("5) Sair\n");
        printf("Opcao: ");
        if (scanf("%d", &opc) != 1) {
             limparBufferEntrada();
             opc = 0; }

        limparBufferEntrada();

        switch (opc) {
            case 1:
```

```
            inserirTodas(corridas, n);
            preenchidas = 1;
            break;

        case 2:
            if (!preenchidas) {
                printf("Insira as corridas
                       primeiro.\n");
                break; }
            listarTodas(corridas, n);
            break;

        case 3:
            if (!preenchidas) {
                printf("Insira as corridas
                       primeiro.\n"); break; }
            {
                Corrida best = melhorTempo
                              (corridas, n);
                // por valor
                printf("\nMelhor tempo:\n");
                imprimirCorrida(best);
            }
            break;

        case 4:
            if (!preenchidas) {
               printf("Insira as corridas
                       primeiro.\n");
                break; }
            alterarTempo(corridas, n);
               // referência
            break;

        case 5:
            printf("A terminar ...\n");
            break;

        default:
            printf("Opcao invalida.\n");
    }
} while (opc != 5);
```

```
    free(corridas);
    return 0;
}
```

15.10 Capítulo 13

13.1.

```
#include <stdio.h>
#include <string.h>

void remover_nova_linha(char *s) {
    size_t n = strlen(s);
    if (n > 0 && s[n-1] == '\n') s[n-1] = '\0';
}

void gravarSeries(const char *ficheiro) {
    FILE *fp = fopen(ficheiro, "w");
    if (!fp) {
        perror("Impossivel abrir para escrita");
        return; }

    char serie[128];
    for (int i = 1; i <= 5; i++) {
        printf("Serie %d: ", i);
        if (!fgets(serie, sizeof(serie), stdin)) {
            i--;
            continue; }
        remover_nova_linha(serie);
        fprintf(fp, "%s\n", serie);
    }
    fclose(fp);
}

void lerSeries(const char *ficheiro) {
    FILE *fp = fopen(ficheiro, "r");
    if (!fp) {
          perror("Impossivel abrir para leitura");
          return; }

    char linha[128];
    int idx = 1;
```

```
    printf("\n=== Lista de Series ===\n");
    while (fgets(linha, sizeof(linha), fp)) {
        remover_nova_linha(linha);
        printf("%2d) %s\n", idx++, linha);
    }
    fclose(fp);
}

int main() {
    const char *ficheiro = "series.txt";
    printf("Introduza 5 series (uma por linha):\n");
    gravarSeries(ficheiro);
    lerSeries(ficheiro);
    return 0;
}
```

13.2.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char nome[51];
    int pontuacao; // 0..10
} Jogo;

void remover_nova_linha(char *s) {
    size_t n = strlen(s);
    if (n > 0 && s[n-1] == '\n') s[n-1] = '\0';
}

void gravarJogos(const char *ficheiro, Jogo jogos[],
                 int n) {
    FILE *fp = fopen(ficheiro, "wb");
    if (!fp) {
        perror("Erro a abrir para escrita"); return; }
    fwrite(jogos, sizeof(Jogo), n, fp);
    fclose(fp);
}
```

```
void lerJogos(const char *ficheiro, Jogo jogos[],int n)
{
    FILE *fp = fopen(ficheiro, "rb");
    if (!fp) {
      perror("Erro a abrir para leitura");
      return; }
    fread(jogos, sizeof(Jogo), n, fp);
    fclose(fp);
}

int compararJogos(const void *a, const void *b) {
    const Jogo *ja = (const Jogo*)a;
    const Jogo *jb = (const Jogo*)b;
    // ordem decrescente por pontuacao;
    // em empate, alfabetica
    if (jb->pontuacao != ja->pontuacao)
        return jb->pontuacao - ja->pontuacao;
    return strcmp(ja->nome, jb->nome);
}

int main() {
    const char *ficheiro = "jogos.bin";
    Jogo jogos[5];

    printf("Introduza 5 jogos (nome e pontuacao
           0..10):\n");
    for (int i = 0; i < 5; i++) {
        char tmp[128];
        printf("Jogo %d - nome: ", i+1);
        fgets(tmp, sizeof(tmp), stdin);
        if (tmp[0]=='\n') { i--; continue; }
        remover_nova_linha(tmp);
        strncpy(jogos[i].nome, tmp, 50);
        jogos[i].nome[50] = '\0';

        int ok = 0;
        while (!ok) {
            printf("Jogo %d - pontuacao (0..10): ",
                   i+1);
            if (scanf("%d", &jogos[i].pontuacao) == 1
                && jogos[i].pontuacao >= 0 &&
                jogos[i].pontuacao <= 10)
```

```
                ok = 1;
            else
                printf("Valor invalido.\n");
            // limpar newline pendente
            int c;
           while ((c = getchar()) != '\n' && c != EOF)
        }
    }

    gravarJogos(ficheiro, jogos, 5);

    // ler de volta, ordenar e mostrar
    Jogo lidos[5];
    lerJogos(ficheiro, lidos, 5);
    qsort(lidos, 5, sizeof(Jogo), compararJogos);

    printf("\n=== Ranking ===\n");
    for (int i = 0; i < 5; i++) {
        printf("%d. %-30s  %2d\n", i+1,
               lidos[i].nome, lidos[i].pontuacao);
    }
    return 0;
}
```

13.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char nome[50];
    char telefone[15];
} Contacto;

int lerContactosTexto(const char *ficheiroTexto,
    Contacto contactos[], int max) {
    FILE *fp = fopen(ficheiroTexto, "r");
    if (!fp) {
        perror("Impossivel abrir o ficheiro de texto");
        return -1; }
```

```
    char linha[256];
    int n = 0;
    while (n < max && fgets(linha, sizeof(linha), fp))
{
        char nome[50], tel[15];
        if (sscanf(linha,"%49[^;];%14s",nome,tel) == 2)
{
            strncpy(contactos[n].nome, nome,
                    sizeof(contactos[n].nome));
           contactos[n].nome[sizeof(contactos[n].nome)-
                             1] = '\0';
            strncpy(contactos[n].telefone, tel,
                    sizeof(contactos[n].telefone));
          contactos[n].telefone[sizeof
                    (contactos[n].telefone)-1] = '\0';
            n++;
        }
    }
    fclose(fp);
    return n;
}

void gravarContactosBinario(const char *ficheiroBin,
Contacto contactos[], int n) {
    FILE *fp = fopen(ficheiroBin, "wb");
    if (!fp) {
         perror("Erro a abrir binario para escrita");
         return; }
    fwrite(contactos, sizeof(Contacto), n, fp);
    fclose(fp);
}

void lerContactosBinario(const char *ficheiroBin,
Contacto contactos[], int n) {
    FILE *fp = fopen(ficheiroBin, "rb");
    if (!fp) {
      perror("Erro a abrir binario para leitura");
      return; }
    fread(contactos, sizeof(Contacto), n, fp);
    fclose(fp);
}
```

```
int main() {
    const char *txt = "contactos.txt";
    const char *bin = "contactos.dat";

    Contacto contactos[200];
    int n = lerContactosTexto(txt, contactos, 200);
    if (n <= 0) {
        printf("Nenhum contacto lido.\n");
        return 0; }

    gravarContactosBinario(bin, contactos, n);

    Contacto lidos[200];
    lerContactosBinario(bin, lidos, n);

    printf("\n=== Contactos (do binario) ===\n");
    for (int i = 0; i < n; i++) {
        printf("%2d) %-30s  %s\n", i+1, lidos[i].nome,
               lidos[i].telefone);
    }
    return 0;
}
```

13.4.

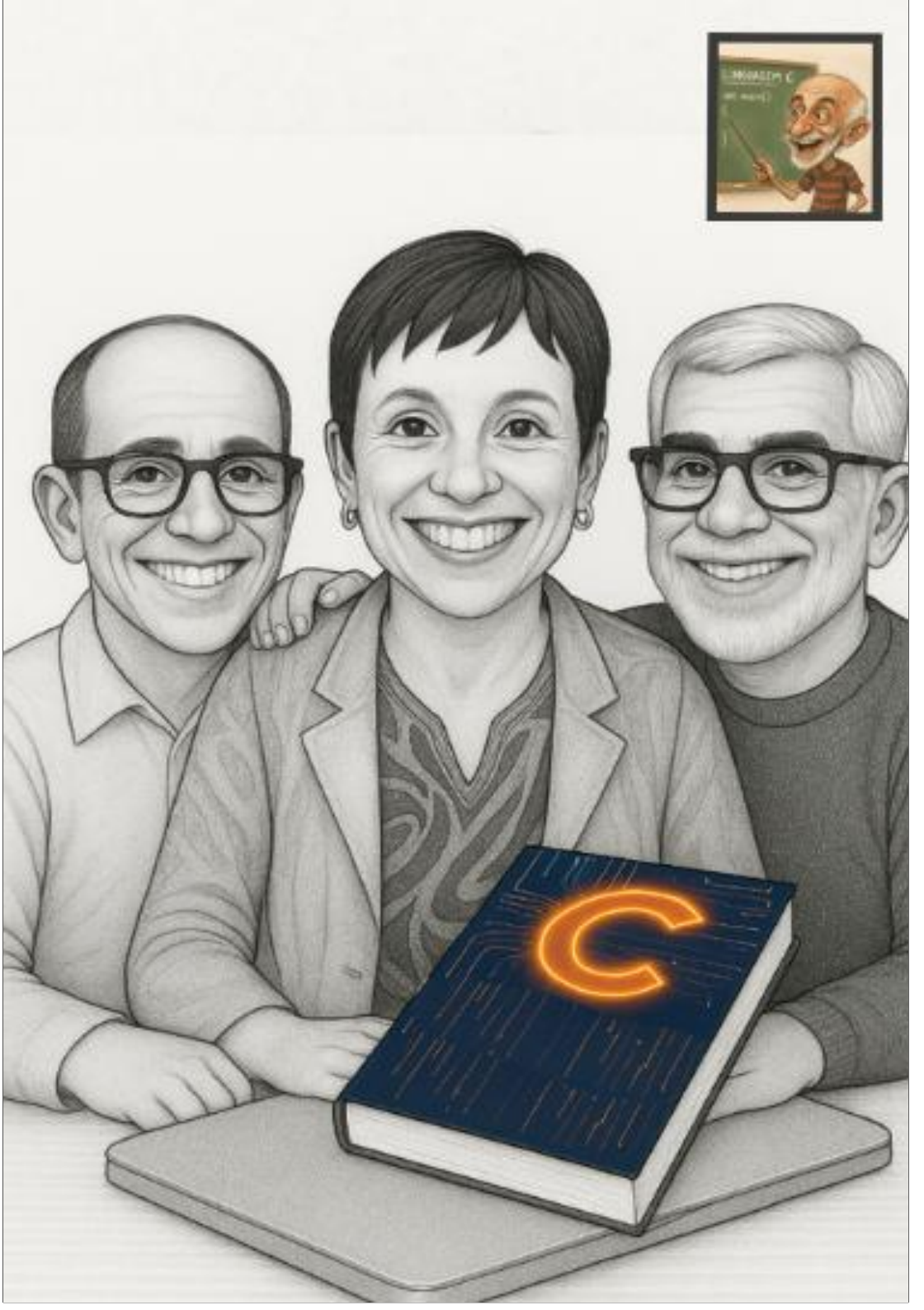
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char produto[50];
    int quantidade;
    float preco;
} Venda;

int lerVendasBinario(const char *ficheiro, Venda
vendas[], int max) {
    FILE *fp = fopen(ficheiro, "rb");
    if (!fp) {
        perror("Nao foi possivel abrir vendas.dat");
        return -1; }
```

```
    int n = 0;
    while (n < max && fread(&vendas[n], sizeof(Venda),
            1, fp) == 1) {
        n++;
    }
    fclose(fp);
    return n;
}

void gerarRelatorioTexto(const char *ficheiroTexto,
                         Venda vendas[], int n) {
    FILE *fp = fopen(ficheiroTexto, "w");
    if (!fp) {
       perror("Nao foi possivel criar relatorio.txt");
       return; }

    fprintf(fp, "%-20s %5s %10s %10s\n", "Produto",
           "Qtd", "Preco", "Total");
    fprintf(fp, "--------------------------------\n");

    for (int i = 0; i < n; i++) {
        float total = vendas[i].quantidade *
                      vendas[i].preco;
        fprintf(fp, "%-20s %5d %10.2f %10.2f\n",
              vendas[i].produto, vendas[i].quantidade,
              vendas[i].preco, total);
    }
    fclose(fp);
}

void criarBackupBinario(const char *ficheiroBackup,
                        Venda vendas[], int n) {
    FILE *fp = fopen(ficheiroBackup, "wb");
    if (!fp) {
       perror("Nao foi possivel criar backup");
       return; }
    fwrite(vendas, sizeof(Venda), n, fp);
    fclose(fp);
}

int main() {
    const char *fonte = "vendas.dat";
```

```
    const char *rel = "relatorio.txt";
    const char *bk = "backup_vendas.dat";

    Venda vendas[1000];
    int n = lerVendasBinario(fonte, vendas, 1000);
    if (n <= 0) {
       printf("Nenhum registo lido.\n");
       return 0; }

    gerarRelatorioTexto(rel, vendas, n);
    criarBackupBinario(bk, vendas, n);

    printf("Relatorio gerado em '%s' e backup criado em
           '%s'.\n", rel, bk);
    return 0;
}
```

16 REFERÊNCIAS BIBLIOGRÁFICAS

1. **KERNIGHAN, B. W.; RITCHIE, D. M.** *The C Programming Language*. 2nd ed. Englewood Cliffs: Prentice Hall, 1988. ISBN 978-0131103627.
2. **DEITEL, H. M.; DEITEL, P. J.** *C: Como Programar*. 8. ed. São Paulo: Pearson, 2016. ISBN 978-8543004799.
3. **PRATA, S.** *C Primer Plus*. 6th ed. Boston: Addison-Wesley, 2013. ISBN 978-0321928429.
4. **KING, K. N.** *C Programming: A Modern Approach*. 2nd ed. New York: W. W. Norton & Company, 2008. ISBN 978-0393979503.
5. **SCHILDT, H.** *C: The Complete Reference*. 4th ed. New York: McGraw Hill, 2000. ISBN 978-0072121241.
6. **KOCHAN, S. G.** *Programming in C*. 4th ed. Boston: Pearson, 2014. ISBN 978-0321776419.
7. **REEK, K. A.** *Pointers on C*. Boston: Addison-Wesley, 1998. ISBN 978-0673999863.
8. **HILL, G.** *Modern C*. Raleigh: The Pragmatic Bookshelf, 2020. ISBN 978-1680507225.
9. **PERRY, G.** *C Programming Absolute Beginner's Guide*. 3rd ed. Indianapolis: Que Publishing, 2013. ISBN 978-0789751987.
10. **YASHWANT, K.** *Test Your C Skills*. 5th ed. New Delhi: BPB Publications, 2015. ISBN 978-8183331630.
11. **GADDIS, T.** *Starting Out with C: From Control Structures to Objects*. 9th ed. Boston: Pearson, 2021. ISBN 978-0136905691.

12. **PRADHAN, M.** *Advanced C Programming*. New Delhi: BPB Publications, 2020. ISBN 978-9389845398.
13. **HANSEN, J.** *The Art of C Programming*. CreateSpace, 2016. ISBN 978-1534968114.
14. **VAN DER LINDEN, P.** *Expert C Programming: Deep C Secrets*. Upper Saddle River: Prentice Hall, 1994. ISBN 978-0131774292.
15. **SEN, S.** *Programming in ANSI C*. 8th ed. New Delhi: McGraw Hill Education, 2020. ISBN 978-9389811928.
16. **SUMMERS, M.** *C Programming in Easy Steps*. 6th ed. Warwickshire: In Easy Steps, 2018. ISBN 978-1840787573.
17. **KELLY, S.; PRITCHARD, P.** *C by Example*. 2nd ed. Boston: Que Publishing, 2000. ISBN 978-0789721843.
18. **GOTTFRIED, B. S.** *Programming with C*. 3rd ed. New York: McGraw Hill, 2010. ISBN 978-0071635183.
19. **HANSEN, J.** *C Memory Management*. CreateSpace, 2017. ISBN 978-1543195372.
20. **STEVENS, W. R.** *Advanced Programming in the UNIX Environment*. 3rd ed. Boston: Addison-Wesley, 2013. (Capítulos com programação C). ISBN 978-0321637734.
21. **PIKE, R.; KERNIGHAN, B. W.** *The Practice of Programming*. Reading: Addison-Wesley, 1999. ISBN 978-0201615869.
22. **VOSS, W.** *C for Engineers and Scientists*. Boston: Addison-Wesley, 2014. ISBN 978-0321967053.
23. **TISCHLER, N.** *Mastering Algorithms with C*. Sebastopol: O'Reilly Media, 1999. ISBN 978-1565924536.

24. **QUINLAN, M.** *Practical C Programming*. 3rd ed. Sebastopol: O'Reilly Media, 1997. ISBN 978-1565923065.
25. **SRIVASTAVA, P.** *C in Depth*. 4th ed. New Delhi: BPB Publications, 2018. ISBN 978-9387284328.
26. **VIEIRA, M. A.** *Linguagem C Completa e Descomplicada*. 2. ed. São Paulo: Érica, 2018. ISBN 978-8536530382.
27. **ANDRADE, F. A.** *Aprendendo a Programar em C*. 1. ed. Lisboa: FCA, 2017. ISBN 978-9727228364.
28. **PINTO, J. S.** *Programar em C: Fundamentos e Aplicações*. Porto: Publindústria, 2012. ISBN 978-9897220066.
29. **DEITEL, H. M.; DEITEL, P. J.** *C++ e C: Fundamentos e Técnicas de Programação*. São Paulo: Pearson, 2014. ISBN 978-8576056484.
30. **MATHUR, A.** *Data Structures Using C*. 3rd ed. New Delhi: PHI Learning, 2017. ISBN 978-8120353272.